Burroughs

B 6700 / B 7700

FORTRAN REFERENCE MANUAL

(RELATIVE TO MARK II.6 RELEASE)

$5.00

THIS MANUAL REPLACES FORM NUMBER 5000458
DATED 12 JULY 1972

Printed in U.S. America

2 SEPTEMBER 1974

5000458

LIST OF EFFECTIVE PAGES

NOTE: Insert latest changed page; dispose of superseded pages.

TOTAL NUMBER OF PAGES IN THIS MANUAL IS 328 CONSISTING OF THE FOLLOWING:

Page No.	Issue
Title	Original
A	Original
i thru xii	Original
1-1 thru 1-5	Original
1-6 blank	Original
2-1 thru 2-3	Original
2-4 blank	Original
3-1 thru 3-5	Original
3-6 blank	Original
4-1 thru 4-13	Original
4-14 blank	Original
5-1 thru 5-5	Original
5-6 blank	Original
6-1 thru 6-8	Original
7-1 thru 7-8	Original
8-1 thru 8-15	Original
8-16 blank	Original
9-1 thru 9-3	Original
9-4 blank	Original
10-1 thru 10-4	Original
11-1 thru 11-16	Original
12-1 thru 12-31	Original
12-32 blank	Original
13-1 thru 13-21	Original
13-22 blank	Original
14-1 thru 14-5	Original
14-6 blank	Original
15-1 thru 15-16	Original
16-1 thru 16-11	Original
16-12 blank	Original
17-1 thru 17-7	Original
17-8 blank	Original
18-1 thru 18-26	Original
19-1 thru 19-28	Original
20-1 thru 20-6	Original
A-1 thru A-4	Original
B-1 thru B-14	Original
C-1 thru C-21	Original
C-22 blank	Original
D-1 thru D-3	Original
D-4 blank	Original
E-1 thru E-9	Original
E-10 blank	Original
1 thru 11	Original
12 blank	Original

COPYRIGHT © 1970, 1972, 1974 BURROUGHS CORPORATION

AA235612 AA235817 AA54483

Burroughs Corporation believes the program described in this manual to be accurate and reliable, and much care has been taken in its preparation. However, the Corporation cannot accept any responsibility, financial or otherwise, for any consequences arising out of the use of this material. The information contained herein is subject to change. Revisions may be issued to advise of such changes and/or additions.

TABLE OF CONTENTS

Chapter		Page
1	INTRODUCTION	1-1
	Guide to Efficient Use	1-1
	B 6700/B 7700 FORTRAN Language	1-2
	Language Description Conventions	1-2
	Basic FORTRAN Concepts	1-4
	Related Documents	1-5
2	PROGRAM STRUCTURE	2-1
	Source Input Format	2-1
	Program Units	2-2
	END Statement	2-2
	Main Program	2-2
	Statement Ordering	2-2
3	CHARACTER SET	3-1
	EBCDIC Input	3-1
	Digits	3-1
	Letters	3-2
	Basic FORTRAN Characters	3-2
	EBCDIC Characters	3-2
	BCL Input	3-3
	Digits	3-3
	Letters	3-3
	Basic FORTRAN Characters	3-3
	BCL Characters	3-3
	BCD Input	3-3
	Digits	3-4
	Letters	3-4
	Basic FORTRAN Characters	3-4
	BCD Characters	3-4
	Character Card Codes	3-4

TABLE OF CONTENTS

Chapter		Page
4	LITERALS	4-1
	Numeric Literals	4-1
	Integer Constants	4-2
	Real Constants	4-2
	Double Precision Constants	4-4
	Complex Constants	4-6
	Octal Constants	4-7
	Hexadecimal Constants	4-8
	Logical Constants	4-10
	Non-Numeric Literals: Strings	4-11
5	VARIABLES	5-1
	Variable Names	5-1
	Array Elements	5-2
6	EXPRESSIONS AND STATEMENTS	6-1
	Expressions	6-1
	Operators	6-1
	Arithmetic Expressions	6-2
	Expression Types	6-3
	Logical Expressions	6-4
	Relational Expressions	6-4
	Construction of Logical Expressions	6-5
	Value Construct	6-6
	Statements	6-6
	Executable Statements	6-6
	Nonexecutable Statements	6-7
	Statement Labels	6-8
7	SPECIFICATION STATEMENTS	7-1
	Type Statements	7-1
	Array Declarators	7-2
	Optional Size Specification	7-3
	Initial Value Lists	7-3
	Examples of Type Statements	7-4
	COMMON Statement	7-4
	COMMON Names	7-4
	Use of Array Declarators	7-4

TABLE OF CONTENTS

Chapter		Page
7	Storage Assignments	7-5
	Examples of COMMON Statements	7-5
	DIMENSION Statement	7-5
	Examples of DIMENSION Statements	7-6
	EQUIVALENCE Statement	7-6
	Examples of EQUIVALENCE Statements	7-7
	EXTERNAL Statement	7-7
	IMPLICIT Statement	7-8
	Initial Character Lists	7-8
	Examples of IMPLICIT Statements	7-8
8	CONTROL STATEMENTS	8-1
	CALL Statement	8-1
	Array Handling	8-2
	Subroutine Returns	8-4
	Examples of CALL Statements	8-4
	CALL EXIT Statement	8-4
	CONTINUE Statement	8-4
	DO Statement	8-5
	Nesting	8-7
	Parameter Alteration	8-7
	GO TO Statement	8-7
	Unconditional GO TO	8-8
	Assigned GO TO	8-8
	Computed GO TO	8-8
	IF Statement	8-9
	Arithmetic IF	8-10
	Logical IF	8-10
	PAUSE Statement	8-11
	RETURN Statement	8-12
	Standard Return	8-12
	Nonstandard Return	8-13
	Example of Nonstandard Return	8-13
	STOP Statement	8-14
	ZIP WITH Statement	8-15

TABLE OF CONTENTS

Chapter		Page
9	ASSIGNMENT STATEMENTS	9-1
	Arithmetic Assignment Statement	9-1
	Logical Assignment Statement	9-2
	GO TO Assignment Statement	9-3
10	DATA STATEMENTS	10-1
	Statement Use	10-1
	Variable Lists	10-1
	The Initial Value List	10-2
	Double Precision Octal and Hexadecimal Constants	10-3
	Assignment Considerations	10-4
	Conversion During Assignment	10-4
11	STORAGE ALLOCATION	11-1
	Simple Variable	11-2
	Integer Variables	11-2
	Real Variables	11-3
	Double Precision Variables	11-4
	Complex Variables	11-5
	Logical Variables	11-6
	Arrays	11-7
	Equivalenced Data Items	11-10
	Single Storage Locations	11-10
	Multiple Storage Locations	11-11
	Array Handling	11-12
	Elements of COMMON Storage	11-14
12	INPUT/OUTPUT STATEMENTS	12-1
	Program Files and File Designators	12-1
	I/O Lists	12-4
	Result Clause Lists	12-5
	Action Specifier	12-5
	Result Descriptor Assignments	12-7
	Examples of Result Clause Lists	12-8
	I/O Statement Syntax	12-8
	READ Statement	12-8
	Files Referenced	12-9
	Record Numbers	12-10

TABLE OF CONTENTS

Chapter		Page
12	Use of Buffers	12-10
	Formatted READ Statement	12-10
	Unformatted READ Statement	12-11
	Internal Data Transfer	12-11
	WRITE Statement	12-12
	File Referenced	12-13
	Record Numbers	12-13
	Formatted WRITE Statement	12-14
	Unformatted WRITE Statement	12-14
	Internal Data Transfer	12-15
	PRINT Statement	12-15
	Files Referenced	12-16
	Record Access	12-17
	Format Considerations	12-17
	PUNCH Statement	12-17
	BACKSPACE Statement	12-18
	CHANGE Statement	12-19
	CLOSE Statement	12-20
	ENDFILE Statement	12-21
	FIND Statement	12-22
	Error Handling	12-23
	INQUIRE Statement	12-23
	LOCK Statement	12-24
	OPEN Statement	12-25
	PURGE Statement	12-26
	REWIND Statement	12-27
	Free Format Input and Output	12-28
	Input	12-28
	Output	12-28
	File Handling Capabilities	12-29
	Line Printer Files	12-29
	Card Reader Files	12-30
	Card Punch Files	12-30
	Disk Files	12-30
	Tape Files	12-30
	Single-File Tape Reels	12-30
	Multi-File Tape Reels	12-31

TABLE OF CONTENTS

Chapter		Page
13	FORMAT SPECIFICATIONS AND STATEMENTS	13-1
	Format Specifications	13-1
	A Format Specification	13-2
	Input Using Aw	13-2
	Output Using Aw	13-2
	C Format Specification	13-3
	D Format Specification	13-4
	Input Using Dw.d	13-4
	Output Using Dw.d	13-5
	E Format Specification	13-5
	Input Using Ew.d	13-5
	Output Using Ew.d	13-6
	F Format Specification	13-6
	Input Using Fw.d	13-6
	Output Using Fw.d	13-7
	G Format Specification	13-7
	H Format Specification (Strings)	13-8
	Input Using wHs or 's'	13-8
	Output Using wHs or 's'	13-9
	I Format Specification	13-9
	Input Using Iw	13-9
	Output Using Iw	13-10
	J Format Specification	13-10
	Input Using Jw	13-10
	Output Using Jw	13-10
	L Format Specification	13-11
	Input Using Lw	13-11
	Output Using Lw	13-11
	O Format Specification	13-12
	Input Using Ow	13-12
	Output Using Ow	13-12
	P Format Specification	13-13
	T Format Specification	13-14
	U Format Specification	13-15
	V Format Specification	13-15
	X Format Specification	13-15
	Z Format Specification	13-15

TABLE OF CONTENTS

Chapter		Page
13	Input Using Zw	13-16
	Output Using Zw	13-16
	Format Specification Modifiers	13-17
	K Format Specification Modifier	13-17
	$ Format Specification Modifier	13-17
	The FORMAT Statement	13-17
	Record Fields	13-18
	Complex I/O List Elements	13-18
	Format Field Separators	13-18
	Repeat Counts	13-19
	Run-Time Format Editing	13-19
	Carriage Control	13-20
	Format Specifications in Arrays	13-21
14	NAMELIST STATEMENTS AND NAMELIST I/O	14-1
	Input Using NAMELIST	14-2
	Output Using NAMELIST	14-5
	Using NAMELIST with Separate Compilations	14-5
15	SUBPROGRAMS AND INTRINSICS	15-1
	Subroutine Subprograms	15-1
	SUBROUTINE Statement	15-1
	SUBROUTINE Names	15-1
	Dummy Argument Lists	15-1
	Use of Subroutines	15-2
	Function Subprograms and Statement Functions	15-3
	Function Subprogram	15-3
	FUNCTION Statement	15-3
	FUNCTION Names	15-4
	Example of Function Subprogram	15-5
	Statement Functions	15-5
	Statement Function Declaration	15-5
	Use of Functions	15-6
	Recursion	15-7
	ENTRY Statement	15-7
	ENTRY Names	15-8
	Dummy Argument Lists	15-8

TABLE OF CONTENTS

Chapter		Page
15	Entry in Subroutine Subprograms	15-8
	Entry in Function Subprograms	15-8
	BLOCK DATA Subprograms	15-9
	BLOCK DATA Statement	15-9
	Use of BLOCK DATA	15-9
	Intrinsics	15-10
	CONCAT Intrinsic	15-13
	CTIME Intrinsic	15-14
	RANDOM Intrinsic	15-14
	TIME Intrinsic	15-15
16	ADDITIONAL LANGUAGE CAPABILITIES	16-1
	Separately-Compiled Program Units	16-1
	Independent Process Initiation	16-4
	Zip With Array	16-5
	Zip With File	16-8
	File Format	16-8
17	DEBUGGING AND TIMING AIDS	17-1
	Introduction	17-1
	DEBUG MONITOR Statement	17-1
	DEBUG DUMP Statement	17-3
	DEBUG PROGRAMDUMP Statement	17-4
	DEBUG STATISTICS Statement	17-5
	DEBUG DUMP STATISTICS Statement	17-6
	DEBUG TRACE Statement	17-6
	Trace List	17-7
18	FORTRAN COMPILER	18-1
	Introduction	18-1
	Installation/Compiler Interface	18-1
	System Requirements	18-1
	Required Hardware	18-1
	Required Software	18-2
	Maintenance of Compiler Software	18-2
	User/Compiler Interface	18-3
	Compiler Files	18-4

TABLE OF CONTENTS

Chapter		Page
18	Input Files	18-6
	Output Files	18-6
	Compiler File Table	18-8
	Sample Card Input Decks	18-12
	Compile for Syntax	18-13
	Compile for Library	18-15
	Compile and Go	18-16
	Compile for Library and Go	18-18
	Batch Facility	18-19
	Deck Set Up	18-19
	Using the Batch Facility	18-20
	Restrictions	18-21
	Compilations Through CANDE	18-22
	Language Compatibility	18-22
	FORTRAN IV, Level H	18-22
	B 3500 FORTRAN	18-24
	ANSI FORTRAN	18-25
	Other FORTRAN Languages	18-25
19	COMPILER OPTIONS	19-1
	Compiler Control Statements	19-1
	Option Actions	19-2
	ASCII (Reset)	19-6
	AUTOBIND (Reset)	19-7
	BCD (Reset)	19-8
	BCL (Reset)	19-8
	BINDER (No associated states)	19-8
	B5500 (Reset)	19-8
	B5700 (Reset)	19-9
	B7700 (Reset)	19-9
	CHARS (No associated states)	19-9
	CHECK (Reset)	19-9
	CODE (Reset)	19-10
	DBLTOSNGL (Reset)	19-10
	DEBUG (Reset)	19-10
	END (No associated states)	19-10
	ENTRY	19-10
	ERRLIST (Reset for non-CANDE compilations, set for CANDE compilations)	19-10

TABLE OF CONTENTS

Chapter		Page
19	FREE (Reset for non-CANDE compilations, set for CANDE compilations)	19-11
	FREENEW (Reset)	19-11
	FREETAPE (Reset)	19-11
	GRAPH (Reset)	19-11
	INCLNEW (Reset)	19-11
	INCLUDE (No associated states)	19-12
	INSTALLATION (Reset)	19-14
	IOTIME (No associated states)	19-15
	JOB (No associated states)	19-15
	LEVEL (No associated states)	19-15
	LIBRARY (Reset)	19-16
	LIMIT (No associated states)	19-16
	LINEINFO (Reset for non-CANDE compilations, set for CANDE compilations)	19-16
	LIST (Set for non-CANDE compilations, reset for CANDE compilations)	19-16
	LONG (Reset)	19-17
	MERGE (Reset)	19-17
	MONITOR (Reset)	19-17
	NEW (Reset)	19-18
	NEWSEGMENT (No associated states)	19-18
	NEWSEQERR (Reset)	19-19
	NOBINDINFO (Reset)	19-19
	NOWARN (Reset)	19-19
	OMIT (Reset)	19-19
	OMITDEBUG (Reset)	19-20
	OPT (No associated states)	19-20
	OWN (Reset)	19-20
	OWNARRAYS (See text)	19-21
	PAGE (No associated states)	19-21
	PRINTER (No associated states)	19-21
	PRINTLIMIT (No associated states)	19-22
	PROCESSTIME (No associated states)	19-22
	READER (No associated states)	19-22
	SEGMENTATION	19-22
	SEPARATE (Reset)	19-22
	SEQ (Reset)	19-23

TABLE OF CONTENTS

Chapter		Page
19	SEQERR (Reset)	19-24
	SINGLE (See text for default value)	19-24
	STACK (Reset)	19-25
	SUPRS (Reset)	19-25
	TIME (Reset)	19-25
	TRACE (Reset)	19-25
	VECTORMODE (Reset)	19-25
	VOID (Reset)	19-26
	VOIDT (Reset)	19-26
	XREF (Reset)	19-26
	$ (Reset)	19-26
	FORTRAN Compiler Generation Options	19-27
	BIGOPTJOB (Reset)	19-27
	FASTGRAPH (Reset, but must be set if OPTLSS1 is reset)	19-27
	GETFIRSTDOLLARCARDFROMSPO (Reset)	19-27
	MODELII (Reset)	19-27
	MULTI (Set)	19-27
	OPTLSS1 (Reset)	19-27
	SETOWNARRAYS (Reset)	19-28
	SINGLEBYDEFAULT (Set)	19-28
	VECTORMODEISALLOWED (Reset)	19-28
20	OPTIMIZATION AND VECTORMODE	20-1
	Optimization	20-1
	OPT = 0	20-1
	OPT = -1	20-1
	OPT = 1	20-2
	Additional Compiler Control Card Considerations	20-3
	Vector Mode	20-4
	APPENDIX A - SAMPLE PROGRAM	A-1
	APPENDIX B - FILE DECLARATION CARDS	B-1
	APPENDIX C - FORTRAN COMPILER ERROR MESSAGES	C-1
	APPENDIX D - FORTRAN FORMAT ERROR MESSAGES	D-1
	APPENDIX E - FORTRAN SYNTAX	E-1

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Number		Page
18-1	FORTRAN Compilation System	18-5

LIST OF TABLES

Number		Page
3-1	Card Codes for Basic FORTRAN Characters	3-5
15-1	B 6700/B 7700 Intrinsics	15-11
18-1	FORTRAN Compiler Files	18-9
B-1	Default Program File Specifications	B-14

CHAPTER 1

INTRODUCTION

GUIDE TO EFFICIENT USE

The purpose of this document is to present information directly connected with the FORTRAN programming language as implemented on the B 6700 and B 7700 data processing systems. Specifically, this document describes both the programming language accepted by the B 6700/B 7700 FORTRAN compiler and the various features of this compiler.

This manual is user-oriented in that it is designed to provide the FORTRAN programmer with a source of reference information pertinent to the manner in which the B 6700/B 7700 system can be used.

The manual is logically divided into two distinct portions. The first portion consists of a discussion of the features of the B 6700/B 7700 FORTRAN language, while the second discusses the B 6700/B 7700 FORTRAN compiler. Chapters 2 through 17 describe the language, and it is to these chapters that the user is referred for programming information. Chapters 18 through 20 are devoted to a description of the compiler and contains information as to the manner in which FORTRAN programs are submitted to the compiler and the extent to which the user may control compiler operation.

Due to the fact that the basic features of the FORTRAN (FORmula TRANslation)[1] language have been standardized and documented elsewhere, this document is concerned primarily with providing information concerning the B 6700/B 7700 implementation of this language. The present manual is not a primer in the language and should not be used as such. The first (and largest) portion of this document is devoted to a description of the constructs of this language as implemented on the B 6700/B 7700 system. The discussion proceeds from basic language elements to general FORTRAN program structures. The pertinent chapters contain the following information:

---

[1] FORTRAN was originally developed for International Business Machine equipment.

INTRODUCTION

- Chapter 2 discusses the form of program statements and program structure.
- Chapters 3 through 5 discuss characters, literals (i.e., constants and strings), and variables, the primary units of which the language is constructed.
- Chapters 6 through 10 discuss FORTRAN expressions and statements, the computation-directed elements of the language.
- Chapter 11 examines the manner in which storage is allocated for program data items.
- Chapters 12 through 14 discuss I/O-related constructs.
- Chapter 15 is concerned with FORTRAN subprograms.
- Chapter 16 contains a discussion of language facilities allowing use of separately-compiled program units and initiation of independent processes.
- Chapter 17 discusses debugging and timing aids.

In addition, a number of appendices are provided at the end of this manual which consist of a number of reference aids for the programmer.

The programmer with questions concerning the legal format of B 6700/B 7700 FORTRAN constructs is directed to the section of this document which discusses the subset of the FORTRAN language containing the type of construct in question.

The programmer with questions concerning the portion of the B 6700/B 7700 system which processes FORTRAN programs (i.e., the FORTRAN compiler) is directed to the portions of chapters 18 and 19 of this document which discuss the compiler feature in question.

Chapter 20 discusses optimization and the use of vectormode hardware.

B 6700/B 7700 FORTRAN LANGUAGE

The B 6700/B 7700 FORTRAN language was designed so that many of the FORTRAN programs already in existence would be compatible with it with minimal changes. In particular, this language was designed to be compatible with FORTRAN IV, H level, and to contain ANSI X3.9-1966 Standard FORTRAN as a subset.

LANGUAGE DESCRIPTION CONVENTIONS

Interspersed throughout the discussion of the FORTRAN language contained in this document are brief indications of the general formats of various language

constructs. These portions of text follow certain conventions to be explained here.

The method employed in this manual is to present the general format of the construct with the optional portions of the construct represented by lower-case letters and defined following the specification of the format. The remaining characters in the specification are required portions of the construct being defined.

The proper format for an Example Item is:

EX(s)MP,LE
where s is a string of from one to three characters, each of which is the character A.

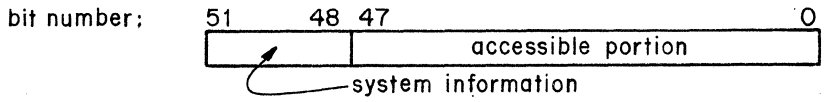
This example consists of the presentation of the general format of a fictitious language construct called an "Example Item." The portion of the construct over which the programmer has control is denoted by the lower-case letter s and is defined below the format specification. The balance of the characters in the example format (i.e., all upper-case letters, the parentheses, and the comma) are required portions of the construct. As defined, valid examples of this sample construct would be:

```
EX(A)MP,LE
EX(A A)MP,LE          (blank characters are ignored in FORTRAN
EX(   AAA)MP,L E      except in string literals)
```

Another convention employed in this document concerns the manner in which internal data words are described. The basic B 6700 data word is 52 bits in length and is described in detail in the B 6700 Hardware Handbook, Form No. 5000276. The term "48 accessible bits" is employed in this manual in relation to data word descriptions; this term refers to the 48 bits of the data word which are accessible to the user. The 52 bits of a data word are numbered 51 to 0, from left to right; the portion of the word which the user may directly access consists of bits 47 through 0.

INTRODUCTION

The B 6700/B 7700 data word is constructed in the following manner:



When it is necessary to illustrate data word fields, this document will display the 48 accessible bits in the following manner:

47	43	39	35	31	27	23	19	15	11	7	3
46	42	38	34	30	26	22	18	14	10	6	2
45	41	37	33	29	25	21	17	13	9	5	1
44	40	36	32	28	24	20	16	12	8	4	0

This data word layout is employed since it facilitates the interpretation of the contents of the data word in terms of 4-bit hexadecimal digits. This is convenient since internal alphabetic processing on this machine is based on 8-bit EBCDIC characters.

BASIC FORTRAN CONCEPTS

Certain basic concepts concerning the FORTRAN language are presented here as preliminary to the following discussion of the B 6700/B 7700 implementation of this language. These concepts are discussed in detail in the following chapters.

A problem-solving procedure written in the FORTRAN language is called a FORTRAN source program.

Each program consists of one or more program units, including a required main program and possible additional subprograms. Some subprograms are provided by the system software and may be referenced by the source program.

Each program unit is constructed of a series of statements. These statements specify the arithmetic operations which are to be executed, control the order in which program statements are to be performed, accomplish various program input and output functions (such as reading data records, printing the results of computations, etc.), describe program data items, or provide other program information without directly producing any actions during program execution.

Each program statement is constructed of a string of appropriate characters which is contained on one or more physical records (e.g., punched cards). The sum of the physical records containing an executable source program constitutes a source deck. This deck may be input as a file to a special computer program called a compiler which processes the input characters, generating an object program which the machine may execute.

The following chapters first discuss the FORTRAN language from the basic level of source characters through the level of complete executable source programs and then describe the operation of the FORTRAN compiler.

RELATED DOCUMENTS

The information in the following Burroughs documents pertains to the material discussed in the present manual.

PUBLICATION	FORM NO.
B 6700 COMMAND AND EDIT (CANDE) LANGUAGE	5000318
B 6700 INPUT/OUTPUT SUBSYSTEM	5000185
B 6700 MATHEMATICAL INTRINSICS	5000151
B 6700 PROGRAM BINDER	5000045
B 6700/B 7700 SYSTEM SOFTWARE HANDBOOK	5000722
B 6700 SYSTEM MISCELLANEA	5000367

CHAPTER 2

PROGRAM STRUCTURE

Every executable FORTRAN program consists of a sequence of statements, with each statement physically contained on one or more lines, or card images.

SOURCE INPUT FORMAT

Unless the compiler option FREE (see chapter 19) is set, FORTRAN statements must be input to the compiler upon cards or card images which have the following format. (All blanks are ignored in source input except when they occur in Hollerith or proper strings.)

Source input cards are in general free form, with these exceptions:

a. Columns 1 through 5 of a card may contain a statement label. (See chapter 6.) This field is recognized as a label on the first card only of an executable or FORMAT statement. On all nonexecutable statements other than the FORMAT statement and on all continuation cards, this field is ignored. A label without an associated statement is ignored. Blanks and preceding zeros are ignored.

b. Column 6 of the first card of a statement must be blank or contain a zero. A statement may be continued on several cards by placing any nonblank and nonzero character in column 6 of the continuation cards. As many continuation cards as necessary may be used.

c. Columns 7 through 72 of a card contain the FORTRAN statement.

d. Columns 73 through 80 are normally not used by the FORTRAN compiler and may contain identification or sequencing information. This field is interpreted as a sequence number when the compiler option MERGE or INCLUDE is set and patches are being merged into a source file. (See chapter 19.)

Two or more executable statements may be put on one card if the statements are separated by a semicolon (;).

If a card contains a percent sign (%) not in a string literal, the remainder of the card (except the sequence number) is ignored and may be used for comments.

PROGRAM STRUCTURE

If a card contains a "C" in column 1, the card is considered a comment card and is not interpreted except for the sequence number in columns 73 through 80 when the MERGE option is set.

A card containing a "$" in column 1 or a blank in column 1 and a "$" in column 2 is a compiler control card as discussed in chapter 19.

Blank characters are significant only in column 6 of a statement card and in string literals. With these exceptions, blanks may be used freely or left out altogether without affecting the meaning of the FORTRAN program.

PROGRAM UNITS

Every executable FORTRAN program consists of exactly one main program unit which may be preceded or followed by as many subprograms as necessary.

Each program unit consists of a sequence of statements terminated by an END statement.

END Statement

The nonexecutable END statement is provided for use as the terminal statement of a program unit.

The proper form of the END statement is:

```
END
```

Every program unit must contain exactly one END statement.

If an END statement is encountered during execution of a subprogram, a RETURN is implied; if an END statement is encountered in a main program, a STOP is implied. An END statement may be continued, but not labeled.

Main Program

The main program is the first (and perhaps only) program unit to be executed.

The main program may invoke SUBROUTINE and FUNCTION subprograms via the CALL statement and function references, respectively. (Subprograms are discussed in chapter 15.) It is recommended that subprograms appear before their first reference, if possible. This will facilitate argument checking at compile-time and may improve efficiency.

Statement Ordering

The order of appearance of statements in the main program or subprogram body is determined by the following rules:

a. File declaration cards must precede the first FORTRAN statement of the program as described in appendix B.

b. An IMPLICIT statement, when used, must be the first statement of the main program or the second statement of a subprogram, except for comment and FILE declaration cards.

c. DIMENSION, COMMON, EQUIVALENCE, and type statements must appear before any affected variables are referenced by a DATA statement.

d. If a DIMENSION, COMMON, or type statement contains any dimension information for an array, then the statement must precede any EQUIVALENCE statement containing that array.

e. FORMAT statements may appear at any point in the program unit.

f. Statement function declarations must precede the first reference to the function.

If optimization is requested by setting the compiler option OPT to a value greater than 0, then the following additional restrictions apply:

a. DIMENSION, COMMON, type, and EQUIVALENCE statements must precede an executable statement referencing any of the affected variables.

b. An EXTERNAL statement must precede a function or subroutine reference passing the affected subprogram name.

Thus, the recommended order in a program unit is as follows:

- FILE declaration cards.
- SUBROUTINE, FUNCTION, or BLOCK DATA statement, if appropriate.
- IMPLICIT statement.
- DIMENSION, COMMON, or type statements in any order.
- EQUIVALENCE statements.
- DATA statements.
- Statement function declarations.
- Remainder of program unit.
- END statement.

An example of a B 6700/B 7700 FORTRAN program is given in appendix A.

CHAPTER 3

CHARACTER SET

Characters are the elements of which a language is constructed. The B 6700/B 7700 FORTRAN language is based upon a prescribed character set which is described in the present chapter.

The B 6700/B 7700 FORTRAN compiler accepts source input coded in EBCDIC, BCL, or BCD characters, though internal processing is carried out in EBCDIC. This discussion will concern itself with each of these input formats.

EBCDIC INPUT

For EBCDIC input, the B 6700/B 7700 FORTRAN character set may be described as consisting of these types of characters:

a. Digits
   (1) Octal digits
   (2) Decimal digits
   (3) Hexadecimal digits
b. Letters
c. Basic FORTRAN characters
d. EBCDIC characters

Various of these character type sets overlap as will be seen here.

Digits

Three types of digits are employed in B 6700/B 7700 FORTRAN: octal digits, decimal digits, and hexadecimal digits.

Octal digits are defined as consisting of characters 0, 1, 2, 3, 4, 5, 6, and 7. These digits are generally used to define program values in terms of the octal (radix 8) number system.

Decimal digits are defined as consisting of the characters in the octal digit set plus the characters 8 and 9. These digits are generally used to define

program values in terms of the decimal (radix 10) number system, and when the term "digit" is used in this manual it will refer to a member of the set of decimal digits.

Hexadecimal digits are defined as consisting of the characters in the decimal digit set plus the characters A, B, C, D, E, F. These digits are generally used to define program values in terms of the hexadecimal (radix 16) number system, where A is equivalent to 10 in the decimal system, B is equivalent to 11 in the decimal system, etc.

These three digit types are used to represent numerical values.

Letters

For the B 6700/B 7700 FORTRAN language, letters are defined as consisting of these 27 characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z $

The dollar sign ($) is included in the letter set (which is often referred to as the set of alphabetic characters) since it may be employed anywhere any other letter may be used.

Basic FORTRAN Characters

The basic FORTRAN characters are defined as consisting of all digit and letter characters plus the characters:

" . , = + - * /) (& ' blank

These characters are called the basic FORTRAN characters since they are the characters of which the FORTRAN constructs are built.

EBCDIC Characters

The EBCDIC (Extended Binary Coded Decimal Interchange Code) characters are defined as consisting of the characters in the basic FORTRAN character set plus the other characters in the standard EBCDIC character set.

BCL INPUT

For BCL input, the B 6700/B 7700 FORTRAN character set may be described as consisting of these types of characters:

a. Digits

   (1) Octal digits
   (2) Decimal digits
   (3) Hexadecimal digits

b. Letters
c. Basic FORTRAN characters
d. BCL characters

Various of these character type sets overlap as will be seen here.

Digits

Digit characters are defined in the same manner as for EBCDIC input.

Letters

Letter characters are defined in the same manner as for EBCDIC input.

Basic FORTRAN Characters

The basic FORTRAN characters are defined in the same manner as for EBCDIC input except that the apostrophe character (') is replaced by the quote mark (").

BCL Characters

The BCL (Burroughs Common Language) characters are defined as consisting of the characters in the basic FORTRAN character set for BCL input plus the characters:

] ; % [@ : ? #

BCD INPUT

For BCD input, the B 6700/B 7700 FORTRAN character set may be described as consisting of these types of characters:

a. Digits

   (1) Octal digits
   (2) Decimal digits
   (3) Hexadecimal digits

CHARACTER SET

b. Letters
c. Basic FORTRAN characters
d. BCD characters

Various of these character type sets overlap as will be seen here.

Digits

Digit characters are defined in the same manner as for EBCDIC input.

Letters

Letter characters are defined in the same manner as for EBCDIC input.

Basic FORTRAN Characters

The basic FORTRAN characters are defined in the same manner as for EBCDIC input excepting that the apostrophe character ('), is replaced by the quote mark (") and the ampersand (&) is not provided.

BCD Characters

The BCD (Binary Coded Decimal) characters (also called Hollerith characters) are defined as consisting of the characters in the basic FORTRAN character set for BCD input plus the characters:

?] ;

CHARACTER CARD CODES

For EBCDIC-, BCL-, and BCD-coded punch card inputs to the FORTRAN compiler, each basic FORTRAN character has a corresponding card code. These character card codes are given in table 3-1. The codes for digit and letter characters (except $) are not presented in this table, since the codes involved are identical for EBCDIC, BCL, and BCD card formats.

Table 3-1. Card Codes for Basic FORTRAN Characters

BASIC FORTRAN CHARACTER	CHARACTER NAME	CARD ROWS PUNCHED		
		EBCDIC	BCL	BCD
=	Replacement Operator	8,6	8,5,0	8,3
+	Plus	12,8,6	12,0	12
-	Minus	11	11	11
*	Asterisk	11,8,4	11,8,4	11,8,4
/	Slash	1,0	1,0	1,0
(	Left Parenthesis	12,8,5	12,8,5	8,4,0
)	Right Parenthesis	11,8,5	11,8,5	12,8,4
,	Comma	8,3,0	8,3,0	8,3,0
.	Decimal Point (period)	12,8,3	12,8,3	12,8,3
'	Apostrophe	8,5	N	N
"	Quote Mark	7,8	8,7,0	8,4
&	Ampersand	12	12	N
$	Dollar Sign	11,8,3	11,8,3	11,8,3
blank	Blank	none	none	none

N = Not a basic FORTRAN character.

CHAPTER 4

LITERALS

The next level of language complexity consists of the literal (numeric constants and strings) and identifier FORTRAN constructs. These items are formed from the basic FORTRAN characters according to prescribed rules. Literals are discussed in this chapter; the various types of identifiers are discussed in later chapters. For example, variables are discussed in the next chapter. Other identifier types include the function name, subroutine name, COMMON block name, and NAMELIST name, each of which is discussed in the chapter concerned with the language feature with which that type of identifier is associated.

Literals function as FORTRAN data items used in problem solving and related operations such as input/output (I/O). The rules governing the use of these constructs are discussed here.

Literals may be divided into "numeric" (constant) and "non-numeric" (string) classifications.

NUMERIC LITERALS

A constant numeric data item may be expressed by a variety of literal representations. These representations are:

a. Integer Constants.
b. Real (Floating-Point) Constants.
c. Double Precision Constants.
d. Complex Constants.
e. Octal Constants.
f. Hexadecimal Constants.
g. Logical Constants.

These seven constant data constructs are discussed in the following paragraphs; their internal storage requirements are discussed in chapter 11.

LITERALS

Integer Constants

An integer constant consists of an optional sign character (+ or -) followed by a string of decimal digits representing an integer not greater than 549755813887.

The proper format of an integer constant is:

sn
where s is an optional sign character (+ or -) and n is a string of decimal digits representing an integer not greater than 549755813887.

The following are examples of valid integer constants:

0	99999999999
+0	03770
-0	1254
17711	-5 7 (interpreted as -57)

The following are examples of invalid integer constants:

1.0	(decimal point not permitted; interpreted as a real constant)
3,000	(no commas or other punctuation permitted)
1548009000000	(maximum value exceeded; interpreted as a double precision constant)

Real Constants

There are two forms for a real constant. The first form consists of an optional sign (+ or -), a string of 11 or fewer significant digits (leading zeros are not significant digits) and a decimal point that must occur before, after or within the decimal digits.

The second form, called the scientific or exponential form, consists of an optional sign (+ or -), a string of decimal digits that may optionally precede, follow, or contain a decimal point, and an exponent part. The exponent part consists of the letter E followed by an integer constant. The value represented is the number preceding the E multiplied by 10 raised to the power denoted by the integer constant following the E. Thus, the real constant 3E2 represents the value 300, which is 3 times 10 to the second power. If the decimal point is omitted, it is assumed to immediately precede the E.

If the number of significant digits exceeds 11, only approximately 11 significant digits will be retained (see chapter 11).

If either form of real constant has no sign, it will be interpreted as representing a positive (or zero) value.

The proper format for a real constant is:

sm.n or smpnEx
where s is an optional sign character (+ or -), m and n are strings of decimal digits with a combined total not exceeding 11 significant digits, one of which (but not both) may be omitted, p is an optional decimal point, and x is an integer constant.

The maximum absolute value of a real constant that is permitted is approximately 4.3135914667E68 = ((8**13)-1)*8**63 and the smallest non-zero absolute value permitted is approximately 8.7581154021E-47 = 8**(-51). Zero is also permitted.

The following are examples of valid real constants:

3.14159265359	2.5E+07 (same as 25000000)	
0.	1.023342E+8	
0.0	23456789012E-22	
2E3)	0000000000007.	
2.E3) (equivalent)	.5 3E-4 2	(interpreted as .53E-42)
2E+3)	-253.	
.075	-.075	
6.02E23	2.9979E08	

The following are examples of invalid real constants:

-1597	(no decimal point or E portion; interpreted as an integer constant)
6.2E+68	(exceeds maximum size limit)
6.2E-48	(smaller than minimum size limit)
2.5E107	(number too large)
E22	(exponent part alone not permitted; interpreted as a variable name)
2.7E1.2	(exponent part must be an integer)
1E2E3	(only one E portion allowed per constant)
2,765,987.	(no commas or other punctuation, except decimal point, permitted)

The internal machine representation of a real constant is essentially similar to the internal representation of an integer constant. Thus, real and integer constants are often interchangeable in application programs; however, care should be taken when intermingling such constants in expressions as the type of constants employed will often alter the outcome of the evaluation of such expressions. (See chapter 9, ASSIGNMENT STATEMENTS.)

Double Precision Constants

There are three forms for a double precision constant. The first form is an integer constant whose absolute value is greater than 549755813887. The second form is the same as the first form of a real constant (no exponent part) except that it must contain a decimal point and 12 or more significant digits. The third form is the same as the exponential or scientific form for a real constant except that the letter D is used in place of the letter E.

The proper format for a double precision constant is:

si or sm.n or supvDx
where s is an optional sign character, i is a string of decimal digits representing a number greater than 549755813887, m and n are strings of decimal digits with a combined total of more than 11 significant digits, one of which (but not both) may be omitted, u and v are strings of decimal digits, one of which (but not both) may be omitted, p is an optional decimal point, and x is an integer constant.

If the number of significant digits in a double precision constant exceeds 22, only approximately 22 significant digits will be retained (see chapter 11).

The maximum absolute value of a double precision constant that is permitted is approximately 1.94882838205028079124469 D+29603 = ((8**26)-1) *8**32754 and the smallest non-zero absolute value permitted is approximately 1.9385458571375858335564 D-29581 = 8**(-32755). Zero is also permitted.

The following are examples of valid double precision constants:

```
3.1415926535897932
3.1415926535897932D0      } (equivalent)
3.1415926535897932D-0

1D3
+1D+003
+1.D+3                    } (equivalent)
1.D0003

1234567890.1234567890123 D 29594
1  2D-2 4259  (interpreted as 12D-24259)
6.63D-034
9.80665D+0
2.7182818284590452353603
27182818284590452353603
```

The following are examples of invalid double precision constants:

```
3.14159265              (less than 12 digits and no D portion;
                        interpreted as a real constant)
2.7 D29604              (exceeds maximum size limit)
2.7 D-29578             (smaller than minimum size limit)
444                     (smaller than 549755813887; interpreted as in-
                           teger constant)
1,234,567,890,123.      (commas not permitted)
1.3E456                 (no D in exponent part)
```

The internal machine representation of a double precision constant requires two consecutive storage locations. Thus, such constants may not be used interchangeably with real or integer constants, but obey rules discussed later in this manual.

LITERALS

Complex Constants

Complex values may be represented in FORTRAN through the use of complex constant literals. A complex constant consists of a "real" part and an "imaginary" part (in that order) separated by a comma character and enclosed within parentheses. Each of these parts may be either an integer or real constant and must obey the particular rules governing that literal type (such as size limits, etc.).

The proper format of a complex constant is:

(m,n)
where m and n are the real and imaginary part, respectively, each of which may be an integer or real constant.

The complex constant (m,n) represents the quantity m+ni, where $i = \sqrt{-1.}$

The following are examples of valid complex constants with the equivalent mathematical expressions beside them:

COMPLEX CONSTANT	EQUIVALENT MATHEMATICAL EXPRESSION
(123,456)	123 + 456i
(3.14159265,1597)	3.14159265 + 1597i
(0,.05)	0 + 0.05i (or 0.05i)
(100.,100.00)	100 + 100i
(1E2,0)	100 + 0i (or 100)
(0,-1)	0 + (-i) (or -i)

The following are examples of invalid complex constants:

(3,)	(missing imaginary part)
(12345678901234.,0)	(neither part may be a double precision constant)
(12.0,72.E68)	(imaginary part exceeds size limit for real constants)
(3,4,5)	(excessive number of parts — more than two)

The internal machine representation of a complex constant requires two consecutive words of storage; the "real" part of the literal is placed in the first of the two words, the "imaginary" part is placed in the following word. Thus, the handling of complex constants follows special rules to be discussed later in this document.

Octal Constants

In some contexts, program values may be represented in other than decimal form in B 6700/B 7700 FORTRAN. One of these alternate representations consists of the octal constant literal which corresponds to notation in a number system with a radix of 8.

An octal constant consists of the letter O preceded or followed by an optional sign character (+ or -) followed by one to 16 octal digit characters. As many embedded blanks may be contained in the constant as desired.

The proper format for an octal constant is:

sOm or Osm
where s is an optional sign character (+ or -) and m is a string of one to 16 octal digit characters.

Since an octal constant may contain up to 16 octal digits (each of which represents 3 bits of storage), this constant may specify the contents of all of the 48 accessible bits of the memory word used to store the constant value. When the minus character (-) is used as the sign character, then the mantissa sign bit [46:1] of the word involved is set to the value (0 or 1) opposite that specified by the digits of the constant. When the sign character is + or omitted, then no alteration of the word involved occurs except that specified by the octal digits contained in the constant literal.

For example, the sign bit of a storage word is set to 1 when its contents are specified by the octal constant 0-200; this word then contains the internal representation of the integer constant -128. (See chapter 11, STORAGE ALLOCATION.) Similarly, the sign bit of the word with contents specified by 0-2000000000000000 is 0 since the minus sign in the octal constant returns bit 46 to zero after the 2 in the upper digit position of the constant.

LITERALS

The following are examples of valid octal constants with their decimal equivalents (i.e., equivalent integer constants):

OCTAL CONSTANT	DECIMAL EQUIVALENT
O0	0
O0000000000000100	64
O100	64
O-77	-63
-O77	-63
O+00 01	1

The following are examples of invalid octal constants:

777763	(missing O)
O1234567812	(contains the decimal digit 8 which is illegal for an octal constant)
O00000000000005555	(contains more than 16 digits)
O 333.33	(contains a decimal point which is illegal for an octal constant)

Care should be taken when employing the octal constant literal. This literal may not be used in many places where other types of constant literals may be employed (e.g., in arithmetic expressions).

Hexadecimal Constants

Another alternate representation of program values consists of the hexadecimal constant literal which corresponds to notation in a number system with a radix of 16.

A hexadecimal constant consists of the letter Z preceded or followed by an optional sign character (+ or -) followed by one to 12 hexadecimal digit characters.

The proper format for a hexadecimal constant is:

sZn or Zsn
where s is an optional sign character (+ or -) and n is a string of one to 12 hexadecimal characters.

The optional sign character performs the same function as it performs for the octal constant, affecting only the mantissa sign bit of the word storing the specified value.

The hexadecimal notation employed on this machine conforms to the standard form whereby each hexadecimal digit corresponds to a unique pattern of four bits within a data word. A list of these 4-bit patterns is given here with the corresponding hexadecimal ("hex") digits denoted:

HEX DIGIT	BIT PATTERN	HEX DIGIT	BIT PATTERN
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Thus, for example, the string of hexadecimal digits 1A2 corresponds to this pattern of twelve data word bits: 000110100010.

The following are examples of valid hexadecimal constants with their decimal equivalents (i.e., equivalent integer constants):

HEXADECIMAL CONSTANT	DECIMAL EQUIVALENT
Z0	0
Z10	16
Z100	256
-Z1000	-4096
Z-10000	-65536
ZF F	255

The following are examples of invalid hexadecimal constants:

FFF60	(missing Z)
Z-ABCDEFG	(contains the character G which is not a hexadecimal digit character)
-Z1002003004 00A0	(contains more than 12 digits)
Z333.33 003	(contains a decimal point which is illegal for a hexadecimal constant)

Care should be taken when employing the optional sign character. Some program values may be represented by eleven or fewer hexadecimal digits. The effect of including a minus sign in a hexadecimal constant representing such values is to denote the negative quantity with the magnitude indicated by the hexadecimal digit characters contained by the literal. (Thus, Z-10 denotes the decimal quantity -16, etc.). This process is altered somewhat in the case of hexadecimal constants containing twelve hexadecimal digits. Such literals may specify the signs of both the mantissa and exponent of a data word as well as specifying their magnitudes. The structure of a B 6700/B 7700 single precision (REAL type) data word is such that bit [45:1] is the exponent sign bit and bit [46:1] is the mantissa sign bit. (See chapter 11, STORAGE ALLOCATION.) When either of these bits is 1, the corresponding quantity is assumed to be negative (e.g., when the exponent sign bit is 1, the exponent is assumed to be a negative value). Thus, the largest value a hexadecimal constant may represent is Z 1FFFFFFFFFFF (or 0.43135914667E69) rather than ZFFFFFFFFFFFF which represents the quantity -0.70064923216E-45 (while Z-FFFFFFFFFFFF represents 0.70064923216E-45).

Care should also be taken as to the location in a program where a hexadecimal constant is employed. This type of literal may not be used in many program constructs where other types of constant literals may be employed (e.g., in arithmetic expressions).

Logical Constants

FORTRAN allows the use of logical operations through the medium of the logical expression. Thus, two logical constant literals are provided to represent the logical values "true" and "false".

These two logical constants have the form:

```
.TRUE.
```

or

```
.FALSE.
```

The use of these logical constants is restricted to certain types of expressions. (See discussion of logical expressions in chapter 6.) The internal machine representations of these two literals are such that the data words corresponding to the constant .TRUE. and the integer constant 1 are identical and the data words corresponding to the constant .FALSE. and the integer constant 0 are identical.

NON-NUMERIC LITERALS: STRINGS

Constant non-numeric data items may be represented in B 6700/B 7700 FORTRAN through the use of the string literal. This literal construct has two different configurations: the Hollerith string and the proper string. These two configurations may be used interchangeably and are distinguishable only by the differences in their formats.

The general format of a Hollerith string is:

wHs
where w is a positive integer constant denoting the width of the string in characters and s is the string in question, being a string of no more than 264 legal EBCDIC, BCL, or BCD characters, depending upon the input character format.

The general format of a proper string is:

's' or "s"
where s is a string of no more than 264 valid EBCDIC characters. For BCL input, this string is delimited by quote marks (") rather than apostrophes (') and must consist of BCL characters.

For EBCDIC input, the Hollerith string may contain any valid EBCDIC character including the apostrophe ('). For BCL and BCD input, the Hollerith string may contain any valid BCL or BCD character, respectively, including the quote mark ("). (See chapter 3, CHARACTER SET.)

For EBCDIC input, the proper string may contain any valid EBCDIC character including the apostrophe (') and quote ("), but these characters are handled in a special manner. If the string delimiter is the apostrophe (quote), then an apostrophe (quote) is interpreted as part of the string only if two adjacent apostrophes (quotes) appear in the proper string in its place; these two apostrophes (quotes) will be treated as representing one such character. Thus, the proper strings "ABC'DE" and 'ABC''DE' each represent the same string of six characters, ABC'DE.

For BCL input, the proper string may contain any valid BCL character, respectively, including the quote mark ("), but this character is handled in a special manner. A quote mark is interpreted as part of the string only if two adjacent quote marks appear in the proper string in its place; these two quote marks will be interpreted as representing one such character. Thus, the proper string "ABC""DE" is interpreted as representing the quantity: ABC"DE, with the two adjacent quote marks being interpreted as one such character.

In all cases, any blank characters within the character string will be considered part of the string (i.e., embedded blanks are not ignored in this context).

The following are examples of strings valid for EBCDIC input (b represents a blank character):

```
2HbQ
4H"C:"
4H'C:'
'bSIDURIbbb5'
020HHE SAID, "DON'T GO."   }
'HE SAID, "DON''T GO."'    } (equivalent)
"HE SAID, ""DON'T GO."""   }
'"b" IS A BLANK'
'%%%%6%%%%'
"ABCDIFGHJKL"
```

The following are examples of strings invalid for EBCDIC input:

3HbbbAB	(character string may not be longer than integer preceding the H indicates)
-6HABCDEF	(string width specification may not be a negative integer)
0HA	(string width specification may not be zero)
'POIU'b'YT'	(this construct will be interpreted as two separate strings rather than one string containing an apostrophe since the two inner apostrophes are not immediately adjacent)

The following are examples of strings valid for BCL input (b represents a blank character):

```
2HbQ
4H"C:"
"bSIDURIbbb5"
"bb""ABCD"   } (equivalent)
7Hbb"ABCD    }
001Hb        } (equivalent)
"b"          }
"(b) IS A BLANK"
"(MSG)  HERE"
```

The following are examples of strings invalid for BCL input:

3HbbbAB	(character string may not be longer than the integer preceding the H indicates)
-1HW	(string width specification may not be a negative number)
0HQ	(string width specification may not be zero)
"ABC"EFGH"	(a quote mark in a BCL string is represented by a double quote)
"POIU"b"YT"	(this construct will be interpreted as two separate strings rather than one string containing a quote mark since the two inner quote marks are not immediately adjacent)

String literals are employed primarily in two ways: as expressions representing the value of a data item and as an input/output (I/O) specification in a FORMAT statement. (Refer to chapter 13.) In no case may a string literal exceed 264 characters in size. The size limit is reduced to six internal characters (12 for double precision) when the value held by a data word is being specified by the literal. This restriction is due to the fact that the B 6700 stores characters internally in EBCDIC format which requires eight memory bits per character, resulting in a word capacity of six characters, filling the 48 accessible bits of a memory word.

CHAPTER 5

VARIABLES

FORTRAN variables, like literals, are symbolic names which are constructed from the FORTRAN character set according to appropriate rules and which represent unique values. Variables, however, represent values which may be altered during program execution.

These constructs are used to identify one or more storage locations for purposes of data storage and retrieval. The contents of these storage locations are accessed by referencing the associated variable name.

This chapter contains a discussion of variable name construction which extends to array names and array elements.

The internal handling of variables is discussed in chapter 11.

VARIABLE NAMES

A FORTRAN variable name is an identifier which consists of a string of one to six alphanumeric characters (i.e., letter or digit characters), with the leading character being a letter (including $). If the variable name is more than six characters long, then only the first six characters of this identifier will be retained internally. Embedded blanks are acceptable and removed by the system.

Variables may be classified into five fundamental types:

a. Integer
b. Real
c. Double Precision
d. Complex
e. Logical

The value represented by a variable of each of these types may be expressed by a constant literal of the same type. Thus, the value represented by an integer variable may be expressed by an integer constant, the value represented by a complex variable may be expressed by a complex constant, and so forth. The values represented by each variable type must, therefore,

VARIABLES

obey the magnitude and significant-digit restrictions governing the corresponding type of constant literal.

Unless declared otherwise in a specific type statement or an IMPLICIT statement (see chapter 7), the identifier will be assigned a type according to its initial character. If this initial character is the letter I, J, K, L, M, or N, then the variable will by default be of INTEGER type. If this initial character is any other letter (including $), then the variable will by default be of REAL type. No such defaults exist for double precision, complex, or logical variables; variables of these types must be declared as such by appropriate type statements or IMPLICIT statements.

The following are examples of valid variable names with the type assigned to them when they are not affected by explicit type or IMPLICIT statements in the program in which the identifiers appear:

AMK599	Type REAL
IF	Type INTEGER (Note that this variable name is valid since there are no reserved words in B 6700/B 7700 FORTRAN.)
$64000	Type REAL
END$Q	Type REAL
OF TEN	Type REAL; interpreted as OFTEN
LOOP3	Type INTEGER
REALNUMBER	Type REAL; truncated to six characters and stored internally as REALNU.
ZIP WITH	Type REAL; truncated and stored as ZIPWIT

The following are examples of invalid variable names:

3LOOP	Variable name may not begin with a digit character
BE-GIN	Characters other than letters, digits, or blanks are not allowed in a variable name
$6,000	Comma character illegally embedded in variable name

ARRAY ELEMENTS

FORTRAN variables may be divided into simple variables which are denoted by a variable name only and array elements which are denoted by an array name followed by a subscript list enclosed in parentheses. The latter variable form indicates membership in a data grouping called an array; this form is discussed here.

An array is an ordered data set corresponding to an n-dimensional organization such that each member may be referenced by an array element with each of the n subscripts in the element denoting location in the appropriate dimension. In B 6700/B 7700 FORTRAN a data array is limited to a maximum of 31 such dimensions.

An array is referenced by a single identifier (the array name) which must have the same form as a simple variable name. This identifier represents the data items contained in the array, all of which are identical in type. This type is indicated by the array name following the same rules for type assignment governing variable names.

Each member of an array is called an array element. The proper format for an array element is:

a (s)
where a is a simple variable name, formed in accordance with rules governing such constructs, and s is the subscript list which consists of as many arithmetic expressions (subscripts) as there are array dimensions (or possibly one in an EQUIVALENCE statement), separated by commas.

A variable name is designated as an array name via an appropriate array declarator in a DIMENSION, type, or COMMON statement. (See chapter 7, SPECIFICATION STATEMENTS.) This declarator is used to declare the maximum number and size of dimensions allowable for the array and must precede the first appearance of the array name in either an executable or DATA statement. In a given program unit an identifier may be used as a simple variable name or an array name but not both. Whenever an array name appears in a program, this array name must be immediately followed by a subscript list, except when the array name appears in:

a. The dummy argument list of a subprogram

b. The actual argument list of a subprogram reference

c. The list of an I/O statement

d. A COMMON, DATA, EQUIVALENCE, or type statement

e. The place of a FORMAT statement label in a READ, WRITE, PRINT, or PUNCH statement.

VARIABLES

Each member of this array may be referenced via an array element with appropriate subscripts. Each arithmetic expression in the subscript list of this construct is evaluated to obtain integer subscript values. A subscript may be any arithmetic expression of real or integer type. If the subscript type is real, it will be rounded to the nearest integer value. (See chapter 9, ARITHMETIC EXPRESSIONS.) Thus, the array element A(1.0 + 0.1) is interpreted as A(1). Normally, the minimum value any subscript may represent is 1 and the maximum value is the maximum value specified for that subscript in the array declaration.

In certain circumstances, the value of a subscript in an array reference may fall outside the range declared for the array. The rule is: any subscript is allowed that causes a reference to an element within a COMMON block containing the array or to an element within an array EQUIVALENCEd to the array. Allowable subscript ranges for arrays in the following program provide examples:

```
COMMON /X/ A(10), Q, B(19)
COMMON T, C(10)
DIMENSION D(10)
EQUIVALENCE (C(8),D(3))
CALL S(A(5))
END
SUBROUTINE S(F)
COMMON /X/ R, E(9)
DIMENSION F(99)
RETURN
END
```

ARRAY NAME	ALLOWABLE SUBSCRIPTS
A	1 TO 30
B	-10 TO 19
C	0 TO 15
D	-5 TO 10
E	0 TO 29
F	-3 TO 26

An array element may never contain a different number of subscripts than the array declarator except in an EQUIVALENCE statement.

The following are examples of valid array elements:

B(I)	
AMK 599(6)	(interpreted as AMK599(6))
ZIPWITH(3) ZIP WITH (3)	(interpreted as ZIPWIT(3) when used where an array element may appear)
I5(I5(3))	(the subscript is itself an array element)
ARRAY2 (1,1,1,1)	
A(R*T)	
Q (3.7)	(interpreted as Q(4))
INT(1+9*SIN(I*PI10))	

The following are examples of invalid array elements:

I(I)	(a subscript must be a valid arithmetic expression; an array name does not constitute such an expression)
ARRAY3(0.2)	(The subscript 0.2 is rounded to 0)
3ARRAY(6)	(An array name may not violate the rules governing variable names)

A detailed discussion of the internal handling of FORTRAN arrays is contained in chapter 11, STORAGE ALLOCATION.

CHAPTER 6

EXPRESSIONS AND STATEMENTS

This chapter discusses the manner in which expressions are constructed and the general features of the statements which form the basis of the FORTRAN language.

EXPRESSIONS

The purpose of expressions is to specify equation-oriented rules whereby a unique data value may be obtained, possibly as a result of operations performed on other data values.

An expression is any valid constant, variable, function reference, or any combination of these items separated by appropriate operators and parentheses. The expression represents the value obtained when the indicated operations are performed on the indicated values.

Expressions may be divided into two basic types according to the type of literal which may represent the value of the expression: arithmetic expressions and logical expressions.

Operators

The operators which may be employed by a FORTRAN expression are listed in the following table with the relative precedence assigned to each operator by the compiler.

EXPRESSIONS AND STATEMENTS

OPERATOR	TYPE	RELATIVE PRECEDENCE	FUNCTION REPRESENTED
**	Arithmetic	9	Exponentiation
unary -	Arithmetic	8	Change of sign
/	Arithmetic	7	Division
*	Arithmetic	7	Multiplication
-	Arithmetic	5	Subtraction
+	Arithmetic	5	Addition
.NE.	Relational	4	Not equal to
.GE.	Relational	4	Greater than or equal to
.GT.	Relational	4	Greater than
.IS.	Relational	4	48- or 96-bit identical to
.EQ.	Relational	4	Equal to
.LE.	Relational	4	Less than or equal to
.LT.	Relational	4	Less than
.NOT.	Logical	3	Logical negation
.AND.	Logical	2	Logical conjunction
.OR.	Logical	1	Logical disjunction

The presence of these operators indicates that an arithmetic or logical operation or a logical comparison is to be performed. Operations of equal precedence are performed from left to right, except exponentiation which is carried out from right to left. The unary + operator is ignored. Parentheses may be used to override operator precedence in the usual manner.

Arithmetic Expressions

An arithmetic expression represents a rule for computing a value corresponding to any type of numeric literal except a logical constant.

An arithmetic expression may contain only arithmetic operators, constants (except logical, octal, and hexadecimal constants), variables, function references, array elements, and grouping parentheses. No logical operands are permissible in arithmetic expressions. In general, mixed operand types are permissible, with the exception that exponents may not be of COMPLEX type.

Immediately adjacent operators are not permissible; signs on constants are considered as unary - and unary + operators. Parentheses should be used to avoid adjacent operators. For example:

```
A**(-2)
```

It is also permissible to use proper strings as operands, provided that such strings contain six or fewer characters. Such strings are interpreted as one data word containing EBCDIC-coded characters. If the string has fewer than six characters, it will be stored left-justified within the data word followed by blanks (i.e., hexadecimal 40's). Therefore, the expression 'ABCDEF'+4.7

is valid (for EBCDIC-coded compiler input). The string represents a data word with the hexadecimal value ZC1C2C3C4C5C6, and 4.7 is stored as a real number as explained in chapter 11. The words are added together as two numeric values.

Expression Types

The types of the operands in an arithmetic expression determine the type of the value obtained from the evaluation of the expression. When a complex value is combined with any other type of value in an operation, the result will be of COMPLEX type. If none of the operands in an arithmetic operation is complex and at least one is double precision, the result will be of DOUBLE PRECISION type. If none of the operands in an arithmetic operation is COMPLEX or DOUBLE PRECISION and at least one of them is real, the result will be of REAL type. Only if both of the operands in an arithmetic operation are integer will the result be of INTEGER type.

The following tables illustrate the resultant types of arithmetic operations depending upon the types of the operands and the operator involved. "DOUBLE" indicates DOUBLE PRECISION type.

For the operators +, -, *, and /, the result of the operation will be of the following type:

TYPE OF FIRST OPERAND	TYPE OF SECOND OPERAND			
	INTEGER	REAL	DOUBLE	COMPLEX
INTEGER	INTEGER	REAL	DOUBLE	COMPLEX
REAL	REAL	REAL	DOUBLE	COMPLEX
DOUBLE	DOUBLE	DOUBLE	DOUBLE	COMPLEX
COMPLEX	COMPLEX	COMPLEX	COMPLEX	COMPLEX

For exponentiation (**) the result of the operation will be of the following type:

TYPE OF BASE	TYPE OF EXPONENT			
	INTEGER	REAL	DOUBLE	COMPLEX
INTEGER	INTEGER	REAL	DOUBLE	ILLEGAL
REAL	REAL	REAL	DOUBLE	ILLEGAL
DOUBLE	DOUBLE	DOUBLE	DOUBLE	ILLEGAL
COMPLEX	COMPLEX	COMPLEX	COMPLEX	ILLEGAL

In the case of a division involving two integer operands, the result will be truncated to an integer value. Thus, the expression 3/2 represents the value 1 and (-8)/3 represents the value -2, while the expression 3.0/2 represents the value 1.5.

Examples

The following are examples of valid arithmetic expressions (all entities are non-logical):

```
6
I+6
SIN(3.14159*(-A))+2
AND(A(1),A(2))
20**FCN1(END(1,2),12)
-B*A
-(-P)
```

The following are examples of invalid arithmetic expressions:

```
B**-A              (illegal adjacent operators)
Q,R                (illegal character)
2+A.AND.B          (illegal logical operator)
(A+B)(C+D)         (illegal implied multiplication)
```

Logical Expressions

A logical expression consists of any valid combination of logical primaries, logical operators, and parentheses. A logical primary may be a logical constant, variable, or function reference, a relational expression, or a logical expression in parentheses.

Relational Expressions

A relational expression consists of two arithmetic expressions separated by one of the seven relational operators and represents the value .TRUE. or .FALSE. as the indicated relation is true or false, respectively. The relation .IS. is true only if the data words referenced on either side of the operator are identical (i.e., each 48 or 96 pairs of bits agree).

In the expression (A+B).GT.C, if A+B is less than or equal to C, the relation will be false, and the expression will represent the value .FALSE.. A, B, and C must be non-logical, non-complex variables only. .EQ. and .NE. are permitted with complex items.

If a relational operator is used to compare character strings, the bits representing the character strings are treated as numerical data. This may produce unexpected results if a character (including a space) occupies bit positions used to represent the sign or exponent of numerical data (see chapter 11). Strings will be compared according to the value of the internal code (i.e., EBCDIC) of the symbols of the string. This means, for example, that strings containing only letters will be compared in alphabetical order if the strings are of length 5 or less and have been input via a Cn ($1 \leq n \leq 5$) format (see chapter 13).

Construction of Logical Expressions

A logical expression represents the value obtained when the indicated logical operations are performed on the appropriate logical valued operands.

Logical negation is expressed by the operator .NOT.. Given the value of a logical primary, the operator .NOT. will change the value to its complement. For example, if A is .TRUE., then .NOT.A is .FALSE.. A relational expression following a .NOT. must be parenthesized (e.g., .NOT.(M.EQ.N)).

The .AND. operator is used to express the logical product of two logical expressions. Given A.AND.B, the operation will yield the value .TRUE. if and only if both A and B are .TRUE.. If either or both of A and B is .FALSE., then A.AND.B is .FALSE..

The .OR. operator is used to find the logical sum of any two logical expressions. Given A.OR.B, the operation will yield the value .TRUE. if either A or B or both are .TRUE.. Only if both A and B are .FALSE. will A.OR.B be .FALSE..

Except when appearing on both sides of a relational operator (i.e., in a relational expression), elements of arithmetic expressions (i.e., non-logical constants, variables, function references, and operators) may not appear in a logical expression.

Parentheses may be used to override operator precedence in the usual manner.

EXPRESSIONS AND STATEMENTS

Examples

The following are examples of valid logical expressions (variables A, B, C, D and array L are LOGICAL):

```
A
B2.GT.B3
A.OR.L(3)
A.AND.Q.LE.SIN(R)+3.7
C.AND..NOT.D
.NOT.(A.OR.B).AND..NOT.(Q.IS.R)
I+1.EQ. FCN(I,3,R,A)
```

The following are examples of invalid logical expressions:

A+B	(illegal operator)
AND(Q,R)	(the AND intrinsic returns a real value)
I+1.OR.B+6	(an arithmetic expression is not a logical primary)

Value Construct

A construct is available in FORTRAN which enables the user to obtain the numeric (integer) values corresponding to various file attribute mnemonics. For a list of these mnemonics and their corresponding values, refer to the Input/Output Subsystem Information Manual, form number 5000185. The VALUE construct uses a single parameter which must be a valid attribute mnemonic. The compile returns an in-line value of the same type as the associated attribute; the construct may be used anywhere that such an expression type is valid.

Examples

```
INQUIRE (37,KIND=DEVICE,EXTMODE=XMODE)
IF (DEVICE .EQ. VALUE(DISK)) GO TO 35
IF (XMODE .EQ. VALUE(BCL)) XMODE=VALUE(EBCDIC)
```

STATEMENTS

Every executable FORTRAN program consists of a sequence of statements. These statements may be classified into executable and nonexecutable statements.

Executable Statements

An executable statement is an instruction that will cause action to be taken whenever that statement is reached during the normal FORTRAN sequence of processing.

The B 6700/B 7700 FORTRAN executable statements and the chapters in which they are discussed are:

STATEMENT	CHAPTER
Assignment (including ASSIGN statement)	9
BACKSPACE	12
CALL	8
CALL EXIT	8
CHANGE	12
CLOSE	12
CONTINUE	8
DO	8
ENDFILE	12
FIND	12
GO TO	8
IF	8
INQUIRE	12
LOCK	12
OPEN	12
PAUSE	8
PRINT	12
PUNCH	12
PURGE	12
READ	12
RETURN	8
REWIND	12
STOP	8
WRITE	12
ZIP WITH	16

Nonexecutable Statements

A nonexecutable statement is an instruction which gives information to the compiler regarding storage allocation, data initialization, I/O editing specifications, and program units.

The B 6700/B 7700 FORTRAN nonexecutable statements and the chapters in which they are discussed in the present document are as indicated:

STATEMENT	CHAPTER
BLOCK DATA	15
COMMON	7
DATA	10
DIMENSION	7
END	1
ENTRY	15
EQUIVALENCE	7
EXTERNAL	7
FORMAT	13
FUNCTION	15
IMPLICIT	7
NAMELIST	14
Statement function definition	15
SUBROUTINE	15
Type	7

EXPRESSIONS AND STATEMENTS

Statement Labels

A statement may be optionally labeled so that it may be referred to in a DO statement, a GO TO statement, an IF statement, a READ, WRITE, PRINT, or PUNCH statement, or an actual argument list.

A statement label consists of a one- to five-digit unsigned integer constant. No statement label may appear on more than one statement in the same program unit. A statement label must appear anywhere in the first through fifth character positions of the initial card image of a statement. (See chapter 2.)

Statement labels on any nonexecutable statement except a FORMAT statement are flagged with a warning message by the compiler and are ignored.

CHAPTER 7

SPECIFICATION STATEMENTS

The nonexecutable specification statements are employed to supply compile-time information about program variables pertaining to variable types, initial values, and storage allocation and to allow subprograms to be used as actual arguments.

The available specification statements include the following:

Type statements
COMMON statement
DIMENSION statement
EQUIVALENCE statement
EXTERNAL statement
IMPLICIT statement

These statements are discussed in the following paragraphs in the order listed.

• TYPE STATEMENTS

The available type statements allow the type of a program variable to be explicitly specified for a program unit. Such type specifications override any implicit specifications (i.e., due to the initial character in the symbolic name of the variable).

Program variables may be assigned the following types:

COMPLEX
DOUBLE PRECISION or REAL*8
INTEGER
LOGICAL
REAL

The proper forms of a type statement are:

1. t m 2. t m/v/ ...
where t is one of the type names just listed, m is a list of one or more variables, array names, array declarators, and/or FUNCTION names separated by commas, v is an initial value list, and the ellipsis (...) indicates as many repetitions of the form m/v/ and/or m as desired, with each repetition preceded by a comma.

SPECIFICATION STATEMENTS

A variable is assigned a default type according to the initial character of its name. If the first letter of the variable name is I, J, K, L, M, or N, the variable will be of type INTEGER by default. All other variables will be of type REAL by default. These defaults may be changed via the IMPLICIT statement. (See discussion of the IMPLICIT statement in this chapter.)

Type statements must precede the use of affected variables in DATA statements and in executable statements if OPT >0. If the type statement contains an array declarator, then the type statement must precede any EQUIVALENCE statement referencing that array.

Array Declarators

A type statement may also be used to declare the size and number of dimensions of a program array via an array declarator in the list of data items. An array declarator has the form:

a (i)
where a is an array name and i is a list of dimensions which are single precision constants and/or dummy variables separated by commas.

The array declarator must have constant dimensions if the declarator appears in a specification statement in a main program. Real dimensions are truncated to obtain integer values.

If an array is a dummy argument, the array declarator may contain dimensions which are dummy arguments or elements of common storage in addition to constant dimensions if the declarator appears in a specification statement in a subprogram. Such dimensions are called adjustable dimensions as the size of the dummy array may be specified at the time the subroutine is invoked. (The discussion of the CALL statement in chapter 8 explains array handling in subprograms.)

The number of dimensions in an array declarator specifies the number of subscripts an element of that array must have. The product of the dimensions of an array declarator which is not a dummy argument determines the number of internal storage locations assigned to the array. The number of these locations is equal to the product for REAL, INTEGER, and LOGICAL arrays. Twice this number of locations are assigned to DOUBLE PRECISION (or REAL*8) and COMPLEX arrays. (See chapter 11.)

Array declarators may also be specified in COMMON or DIMENSION statements, but a particular array may have only one declarator per program unit.

Optional Size Specification

The items of the list of a REAL type statement may optionally be followed by the specification: *8. Such items will be assigned the type DOUBLE PRECISION.

For example, the type statement:

```
REAL I, J*8, K
```

will assign the type REAL to I and K and the type DOUBLE PRECISION to J.

Initial Value Lists

The type statement may be used instead of the DATA statement for variable initialization. (See chapter 10.) Type statements may optionally contain initial value lists constructed in the same fashion as the initial value lists in a DATA statement (as shown in type statement form 2).

The default initial value of an arithmetic item is 0 or (0,0). The default initial value of a logical variable is .FALSE..

Any number of simple variables may be initialized in one type statement initial value list. If an array is to be initialized, it must be the last declared item preceding the initial value list, which may not contain more values than the number of elements in the array.

If there are more variables in the type statement than there are values in the initial value list, then the latter portion of the variables will be initialized. Arrays are initialized from the beginning of the array.

For example, when this type statement is encountered during compilation:

```
REAL C, A(3)/1,3HMNP/, INT
```

the variables C and INT will be uninitialized (i.e., will retain their default initial values of zero), the array element A(1) will be initialized to 1, the array element A(2) will be initialized to MNPbbb, and the array element A(3) will retain its default initial value of zero.

This type statement is illegal:

```
REAL X,Y(3)/1,2,3,4/
```

since the initial value list contains more values than there are elements in Y.

This type statement is a valid example:

```
REAL X,Y,Z/1,2/, A(2)/2*3/, B(3)/3*1/
```

In this example, Y and Z will be initialized to 1 and 2, respectively, each element of A will be initialized to 3, each element of B will be initialized to 1, and X will retain its default initial value of zero.

SPECIFICATION STATEMENTS

Examples of Type Statements

```
REAL I,J,JTEST,D*8/1.2D3/
LOGICAL L1,L2/T/
DOUBLE PRECISION D
COMPLEX CC(N,6)
INTEGER A(20,20,4)/1600*1/
REAL*8 DOUBLE, DFCN1, $FCN/2*3.6D20/
```

• COMMON STATEMENT

The nonexecutable COMMON statement associates variables and arrays with blocks of storage which may be shared among several program units.

The proper form of the COMMON statement is:

COMMON /n/v ... or COMMON v ...
where n is an optional COMMON name, v is a list of variables, array names, and/or array declarators, and the ellipsis (...) indicates as many repetitions of the form /n/v as desired.

COMMON Names

A symbolic name is associated with each block of COMMON storage; this name is called a COMMON block name. Any program unit may access the block of storage associated with this name via a COMMON statement employing this name. COMMON storage associated with a COMMON block name is called labeled COMMON.

A COMMON block name is constructed in the same manner as a variable name. This name may not be a dummy argument, and it may not appear anywhere except in a COMMON statement.

A COMMON block need not be named; COMMON storage associated with no name is called blank COMMON and is assigned the internal identifier ".BLNK.". If the specification for blank COMMON is the first specification in a COMMON statement, the two slashes enclosing the COMMON name may be omitted. Thus, these two statements are equivalent:

```
COMMON//A,B(10)
COMMON A,B(10)
```

Use of Array Declarators

Array declarators may be used in COMMON statements to declare the dimensionality of arrays in the same manner as type statements or DIMENSION statements. Array declarators are discussed in this chapter under the heading TYPE STATEMENTS.

Storage Assignments

Variables and arrays are assigned contiguous locations in COMMON storage in the order in which they appear in a COMMON statement. No dummy arguments may appear in a COMMON statement.

Entire program arrays but not individual array elements may be assigned storage locations in COMMON storage. No COMMON block may exceed 65,535 words.

If the same COMMON name appears more than once in a program unit, the common elements associated with a second or subsequent appearance are considered as extensions to the list of the previous appearance.

A discussion of the manner in which locations are allocated to elements of COMMON storage is contained in chapter 11. Data initialization may be performed via a BLOCK DATA program unit or in the main program. (See chapter 15.)

A DOUBLE PRECISION or COMPLEX variable in a COMMON block must be positioned such that the first of the two words containing the variable is always located at an odd-numbered word location relative to the beginning of the block.

Variables and array names may not be duplicated in COMMON statements. One variable may not be assigned to more than one block of a COMMON storage within a program unit.

Examples of COMMON Statements

```
COMMON/BLOCK1/A,B(10),C//G,HOLD/BLOCK2/Q(3)
COMMON C
COMMON T1/CMN/T2,T3
```

• DIMENSION STATEMENT

The nonexecutable DIMENSION statement specifies the size and number of dimensions of a program array.

The proper form of the DIMENSION statement is:

DIMENSION d
where d is a list of array declarators separated by commas.

Each array referenced in a program unit must have its array bounds specified exactly once in that program unit. This specification may be accomplished via a DIMENSION, type, or COMMON statement.

For an array which is not a dummy argument, an array declarator specifies exactly the amount of internal storage to be allocated to the array and the number of subscripts an element of that array must have. (See TYPE STATEMENT.)

Only an array that is a dummy argument in a subprogram may have declared dimensions that are variables. These variables must either be dummy arguments or elements of common storage. The total number of words of storage required by the array may not exceed the size of the actual argument as initially declared with constant dimensions. No array may exceed 65,535 words. Array storage is discussed in chapter 11.

Examples of DIMENSION Statements

```
DIMENSION A(10)
DIMENSION B(N,2),C(6)
DIMENSION Q(J)
```

• EQUIVALENCE STATEMENT

The nonexecutable EQUIVALENCE statement causes two or more variables or arrays referenced in a program unit to share the same memory locations.

The proper form of the EQUIVALENCE statement is:

EQUIVALENCE (k) ...
where k is a list of two or more simple variables, array elements, or array names separated by commas, and the ellipsis (...) indicates as many repetitions of the form (k) as desired, with each repetition preceded by a comma.

No dummy argument may appear in an EQUIVALENCE statement list.

The subscripts of array elements in the list must be integer constants and must either correspond in number to the number of dimensions declared for the array or be one in number. When one subscript of value n is employed, the element referenced is the n-th element of the internal array as determined by the way array elements are assigned storage. (See ARRAYS, chapter 11.) An array name is considered as that name with a subscript of one.

Each data item grouping in the EQUIVALENCE statement is enclosed in parentheses and indicates that the equivalent entities share some or all of the same storage locations. When arrays are involved, the indicated array element denotes the EQUIVALENCEd arrays are to overlap and be aligned in such a manner that the incated elements share a storage location(s). No group may contain elements of the same array. Thus, EQUIVALENCE (A(3),B,A(6)) is invalid.

The EQUIVALENCE statement may be used to associate additional elements with a COMMON block. This may extend the block beyond its former terminal point, thus increasing the size of the COMMON block. An EQUIVALENCE statement may not associate an array with a COMMON block in such a manner as to expand that block backwards to locations preceding the initial location of the block.

Two elements of COMMON storage may not be made equivalent to one another, either directly or indirectly, by an EQUIVALENCE statement.

Appropriate storage considerations are discussed in chapter 11.

An EQUIVALENCE statement must precede any executable statement employing items EQUIVALENCEd.

Examples of EQUIVALENCE Statements

```
EQUIVALENCE (A,B,C),(A(3),R(9),G)
EQUIVALENCE (D,E)
```

• EXTERNAL STATEMENT

When an actual parameter list of a function reference or a subroutine call contains a function or subroutine name, that name must appear in an EXTERNAL statement in the same program unit in which that reference or call appears.

The proper form of the nonexecutable EXTERNAL statement is:

EXTERNAL v
where v is a list of function and subroutine names separated by commas.

The EXTERNAL statement must appear before the function or subroutine reference. A statement function name may not appear in an EXTERNAL statement.

If a CALL or function reference appears in a subroutine or function that passes itself as an actual argument, then the subroutine or function must name itself in an EXTERNAL statement.

Any FORTRAN in-line intrinsic specified in an EXTERNAL statement will not be expanded in the code file. Such intrinsic names will be assumed to denote subprograms in the executable program. (See chapter 15.)

The following are examples of valid EXTERNAL statements:

```
EXTERNAL EQUIV
EXTERNAL SUBA, FCN1
```

SPECIFICATION STATEMENTS

• IMPLICIT STATEMENT

The nonexecutable IMPLICIT statement is an auxiliary type statement which allows the default types assigned to variables due to their initial characters to be altered.

The proper form of the IMPLICIT statement is:

IMPLICIT t(c) ...
where t is a type name (see TYPE STATEMENTS), c is an initial character list, and the ellipsis (...) indicates as many repetitions of the form t(c) as desired, with a comma preceding each repetition.

Only one IMPLICIT statement is allowed in a program unit, and any program unit may contain an IMPLICIT statement. If used, the IMPLICIT statement must be the first statement of the main program or the second statement of a subprogram (except for COMMENT cards). The IMPLICIT applies only to symbolic names in the program unit in which the statement appears, including FUNCTION and ENTRY names and dummy arguments.

Initial Character Lists

The list of initial characters appearing in an IMPLICIT statement is constructed as follows.

This list indicates that symbolic names whose initial character lies between or is the same as one of the indicated letters are to be of the specified type. Each element of the list may be a letter or two letters separated by a hyphen (i.e., the minus sign, -). If the element is a letter, a name must begin with that letter to be assigned the specified default type. If the element is a hyphenated letter pair, then this letter pair indicates a range of initial characters with which the default type is associated.

The second of the letters in a hyphenated pair must follow the first in the alphabet. The $ is assumed to follow Z.

Examples of IMPLICIT Statements

The following are valid IMPLICIT statements:

```
     IMPLICIT REAL(I-N)
     IMPLICIT REAL(A-$)
(The preceding two statements have identical effect.)
     IMPLICIT DOUBLE PRECISION(D)
     IMPLICIT LOGICAL(A-C,L), REAL*8 (D-F), COMPLEX(X)
```

CHAPTER 8

CONTROL STATEMENTS

The executable control statements are used to alter the normal flow of program execution. These statements may transfer control to another part of the program, terminate or suspend execution, or control iterative processes.

The control statements consist of the following:

CALL statement
CALL EXIT statement
CONTINUE statement
DO statement
GO TO statement
IF statement
PAUSE statement
RETURN statement
STOP statement
ZIP WITH statement (See chapter 16.)

These statements are discussed in the following paragraphs in the order just listed.

• CALL STATEMENT

The executable CALL statement causes the specified subroutine to be executed.

The proper forms of the CALL statement are:

CALL s or CALL s (a)
where s is the name of a SUBROUTINE subprogram or an ENTRY name in this executable program and a is a list of actual arguments separated by commas.

The elements of the list of actual arguments must agree in number, order of appearance, and type with the dummy arguments appearing in the SUBROUTINE or the ENTRY statement appearing in the called subprogram. These actual arguments are evaluated in left-to-right order before the subroutine is entered.

If a dummy argument is real, an integer actual argument may be used, and vice versa. Octal and hexadecimal constants may not be used as actual arguments.

An actual argument in a subroutine reference may be one of the following items:

a. A Hollerith or proper string.
b. A simple variable.
c. An array element.
d. An array name.
e. An expression.
f. A FUNCTION or SUBROUTINE name.
g. An item of the form: &n, where n is a statement label.

Execution of a CALL statement results in an association of actual arguments with all appearances of dummy arguments in executable statements in the subroutine body, and in an association of actual arguments with variable dimensions in the subroutine, if any.

Following these associations, control is transferred to the first executable statement following the appropriate SUBROUTINE or ENTRY statement.

If an actual argument is an array element with an arithmetic expression as a subscript, then the arithmetic expression is evaluated at the time of the execution of the CALL statement, and the resulting array element is associated with the corresponding dummy argument in the subroutine.

Array Handling

If a dummy argument of a subroutine is an array name, the corresponding actual argument in the CALL statement must be an array name, an array element, or a Hollerith or proper string. The following correspondence is established between the actual array and the dummy array in the subroutine at the time the CALL statement is executed.

If the actual argument is an array name, the entire array may be accessed in the subroutine. The dummy array may be assigned as many dimensions as desired. The correspondence between the internal storage locations assigned the actual array and the elements of the dummy array will be established as described in chapter 11. The subscripts appearing in the statement declaring the dummy array in the subroutine serve two purposes. They determine the correspondence between actual and dummy array elements; they also determine the number of array elements referenced when the dummy array name appears without subscripts in an I/O list of the subroutine.

For example, if the array A is declared via the statement:

```
REAL A(20)
```

and is passed as an actual argument to the dummy argument B which is declared in the subroutine via the statement:

```
REAL B(5),
```

then all twenty elements of A may be referenced via B (i.e., B(18) is not an invalid array element), but the statement:

```
WRITE(6,1) B
```

will cause only the first five of these elements to be output.

If the actual argument is an array element, the array from that element on may be accessed in the subroutine. Considering the preceding example, if the array element A(11) had been passed to the subroutine instead of A, then the last ten elements of this array (i.e., A(11) through A(20)) could be referenced via the dummy array B. The WRITE statement displayed would then cause the first five of these elements (i.e., A(11) through A(15)) to be output.

The following program illustrates the correspondence between a two-dimensional actual argument array A and a corresponding two-dimensional dummy argument B. The storage locations for B consist of the first 15 storage locations of A as described in chapter 11.

```
      DIMENSION N(4,6)
      DO 2 I=1,4
      DO 1 J=1,6
    1 N(I,J)=10*I+J
    2 PRINT/,(N(I,J),J=1,6)
      CALL S(N)
      END

      SUBROUTINE S(N)
      DIMENSION N(3,5)
      DO 1 I=1,3
    1 PRINT/,(N(I,J),J=1,5)
      RETURN
      END

      11, 12, 13, 14, 15, 16,
      21, 22, 23, 24, 25, 26,
      31, 32, 33, 34, 35, 36,
      41, 42, 43, 44, 45, 46,
      11, 41, 32, 23, 14,
      21, 12, 42, 33, 24,
      31, 22, 13, 43, 34,
```

CONTROL STATEMENTS

Subroutine Returns

If the subroutine does not cause termination of the program, the subprogram may return control to the next executable statement following the CALL statement or to the executable statement whose label appears in the actual argument list preceded by an ampersand (&). (See the discussion of the RETURN statement in this chapter.)

Examples of CALL Statements

The following EBCDIC-coded CALL statement examples illustrate the use of actual arguments.

```
CALL SUBA(A, B(2), 3HABC, 'TITLE', I+20)
CALL DATE(O)
```

• CALL EXIT STATEMENT

The executable CALL EXIT statement is provided to allow the termination of an executing program.

The proper format of the CALL EXIT statement is:

```
CALL EXIT
```

The execution of this statement within a program which does not contain a subprogram named EXIT produces exactly the same result as the execution of a STOP statement.

This statement may appear at any point within any program unit where an executable statement may appear.

• CONTINUE STATEMENT

The executable CONTINUE statement is provided primarily as a program documentation aid as the execution of this statement produces no action.

The proper format of the CONTINUE statement is:

```
CONTINUE
```

The CONTINUE statement is used primarily as a dummy executable statement allowing the programmer to position a label at any desired point within a program. This facilitates transfers to that point and allows the range of a DO loop to be clearly delimited.

Examples of the use of the CONTINUE statement are provided in the following sample portion of a FORTRAN program.

```
      .
      .
      .
      DO 1 I=2,10,2
      A(I)= I/M
      PRINT F,A(I)
      IF (A(I)) 3,1,1
1     CONTINUE
      M=-M
3     CONTINUE
      .
      .
      .
```

In this sample of source code, the CONTINUE statement labeled 1 is used as the final statement of a DO loop, and the CONTINUE statement labeled 3 is used as a transfer point for an arithmetic IF statement. The use of the CONTINUE allows the end of the DO loop to be readily located within a listing of the program. The second CONTINUE statement allows transfer to an arbitrary point in the program, the location of which may be changed merely by changing the location of the CONTINUE card.

• DO STATEMENT

The executable DO statement is a control statement provided to alter the order of the execution of program statements in such a manner that a series of statements will be repeatedly executed while the value of a specified program variable is varied between specified limits. The number of iterations is dependent upon this control variable.

The proper forms of the DO statement are:

DO s v=m,n,i or DO s v=m,n
where s is a label on an executable statement following the DO statement in the same program unit, v is the single precision control variable, and m, n, and i are the initial parameter, terminal parameter, and incrementation parameter, respectively, each of which is an arithmetic expression.

CONTROL STATEMENTS

The DO statement causes repeated execution of the statements in its range. The range of a DO statement is defined to be the first executable statement following the DO up to and including the statement whose label is specified in the DO statement.

The initial parameter, terminal parameter, and incrementation parameter of a DO statement may be any integer or real arithmetic expression. If the optional incrementation parameter is omitted, a value of 1 is assumed by the compiler.

After each execution of the range of the DO, and after the control variable has been incremented by the value of the incrementation parameter, a test is made to determine if the control variable is greater than the terminal parameter. If so, the DO is considered satisfied and control is transferred to the first executable statement after the range of the DO. If not, the range of the DO is executed again. In any case, the range of a DO is executed at least once. If the increment is negative, the range of the DO will be executed exactly once or will be repeatedly executed unless there is a branch out of the loop.

The following program prints out the cosine of the numbers -5.4, -5.3, -5.2, and -5.1.

```
      DO 8 X = -5.4, -5.05, .1
    8 PRINT/, COS(X)
      END
```

The control variable is available for use within its range, and may be modified as desired. The incrementation parameter and terminal parameter are reevaluated prior to their use and may also be modified by any statement within the range. There is no restriction on transfers out of or into the range of a DO. If a transfer is made into the range of a DO, the programmer is responsible for the appropriate assignment of a value to the control variable. When a DO statement has been satisfied, the value of its control variable is the value which failed the final test.

The following is a second example of a section of a program unit using a DO statement:

```
          J= 10
          DO 2 I= 1,J
          A(I)=0
    2     B(I,I)=0
```

In this DO loop, A (1) through A (10) inclusive and B (1,1), B (2,2),..., B (10,10) will all be set to zero. (The FORTRAN compiler, however, initially sets all variables to zero unless otherwise specified, such as in an initial value list or DATA statement. This may reduce the requirement for "zeroing" out arrays.)

Nesting

Any number of DO statements may be nested within the range of another DO statement with the following restriction: if a DO statement occurs in the range of another DO, the range of the former must be completely contained within the range of the latter. Both may, however, specify the same statement as the last statement in their ranges. For example:

```
      DO 10 I=1,23
      DO 10 J=3,I-1
10    C(I,J+1)= I*J
```

When several DO statements share the last statement of their ranges, the control variable of one DO statement is not reassigned and tested until each DO statement in its range is satisfied. When the outer-most DO statement is satisfied, control continues with the next executable statement following its range.

Parameter Alteration

DO loop parameters may be changed during the execution of the DO loop. For example:

```
      K=100
      DO 24 INTGR=2,K,2
      SUM=SUM + INTGR
24    K=K-2
```

In this example, the variable K is diminished in value by 2 each time that the DO loop is executed. This will affect the indexing in such a manner as to cause termination of the DO loop when INTGR=50 rather than 100 as was initially declared. Again observe that SUM is automatically initialized to 0 by the compiler so if this is the first time in the program that SUM is referenced, it need not be initialized if the starting value desired is 0.

The control variable INTGR is incremented by 2 each time the DO loop is executed because of the presence of the incrementation parameter 2.

• GO TO STATEMENT

The executable GO TO statement can be used to transfer control from one point of an executing program to another point in the same program unit.

CONTROL STATEMENTS

The proper forms of the GO TO statement are:

```
1.    GO TO n
2.    GO TO v,(m)
3.    GO TO v
4.    GO TO (m),a
```

where n is a label appearing on an executable statement appearing in the same program unit, v is a simple integer variable, m is a list of statement labels separated by commas, and a is an arithmetic expression. In forms 2 and 4, the comma is optional.

Unconditional GO TO

The simplest form of the GO TO statement is displayed by form 1, which represents the unconditional GO TO statement.

Execution of this control statement causes the executable statement bearing the indicated label in the program unit to be the next statement executed.

For example, the statement GO TO 23 causes the statement labeled 23 to be the next statement executed. Normal program flow will resume with the statement following the target of the GO TO if it is not another control statement.

Assigned GO TO

Forms 2 and 3 represent the assigned GO TO statement. The execution of this statement causes control to be transferred to the statement whose label was last assigned to the simple integer variable v by an ASSIGN statement. (See chapter 9.) The list of labels which may optionally be included as part of this statement may be used as a documentation aid in listing the labels to which transfer may be made. This list is ignored during execution.

Thus, the execution of the statements:

```
ASSIGN 23 TO K
GO TO K
```

will cause control to be transferred to statement number 23.

Failure to assign a statement label to the variable v with an ASSIGN statement will cause program termination when the assigned GO TO is executed.

Computed GO TO

Form 4 represents the computed GO TO statement. The execution of this statement causes control to be transferred to a statement whose label appears in the list portion of the statement or to the next executable statement following the GO TO. How control is transferred depends on the value of the arith-

metic expression following the label list. It is evaluated, truncated if not an integer, and is used to select one of the labels in the list. If the expression has the value n, when computed, control passes to the n-th label in the label list. If there are fewer than n labels in the list or if n is less than or equal to zero, control passes on to the next statement.

An example of a computed GO TO statement is:

```
GO TO (1,25,3,6,1,17), I+1
```

At execution time, the value of I+1 is computed. If I+1 has the value n, then control will pass to the n-th statement in the list. For example, if I+1 = 4 (i.e., I=3), then control will pass to the statement labeled 6, the fourth label in the list. If I+1 = 1 or 5 (I=0 or 4) in this particular example, control will pass to statement number 1, since both the first and fifth elements of the label list are 1. In this example, if I+1 is less than 1 or greater than 6 (I less than 0 or greater than 5), control will pass to the next executable statement after the GO TO.

Notice also that the statement:

```
GO TO (3), I
```

is not the same as:

```
GO TO 3
```

The latter case is unconditional, whereas, in the former case control will pass to statement number 3 only if I=1. If I has any other value, control will pass on to the next statement, which may or may not be statement number 3.

• IF STATEMENT

The executable IF statement is a control statement provided to cause branching from one point in an executing program to another point depending upon an arithmetic or logical value.

The proper forms of the IF statement are:

1. IF(a) n,z,p 2. IF(e) s
where a is an arithmetic expression, n, z, and p are labels appearing on executable statements in the same program unit, e is a logical expression, and s is any executable statement except a DO statement or a form 2 IF statement.

CONTROL STATEMENTS

Arithmetic IF

Form 1 of the IF statement represents an arithmetic IF statement. The arithmetic IF statement is a three-way branch. The arithmetic expression inside the parentheses following the IF is evaluated, and control is transferred to the statement identified by the first, second, or third label, depending on whether the expression is negative, zero, or non-zero positive, respectively.

Since complex numbers are not ordered as negative or positive, the expression in an arithmetic IF may not be of complex type.

An example of an arithmetic IF is:

```
IF (I-J) 3, 986, 17
```

If I-J is negative, control will be transferred to statement number 3; if zero, to statement number 986; and if positive, to statement number 17. Notice that this is actually a test of whether or not J is greater than, equal to, or less than I.

Not all three statement numbers of an arithmetic IF need be different. For example:

```
IF((A-2)*(B-3)) 10, 10, 3
```

Here control will pass to statement number 3 only if (A-2)*(B-3) is greater than zero. Otherwise, control will pass to statement number 10.

Notice that if all three statement numbers in the arithmetic IF statement are identical, the result is the same as an unconditional GO TO, provided that execution is not terminated during evaluation of the expression.

The statement following an arithmetic IF generally will need a label. This is not a syntactical requirement, but since an arithmetic IF breaks the sequential flow of execution to a labeled statement, it will never be possible to return to execute the statement following the IF unless that subsequent statement has a label.

Logical IF

Form 2 of the IF statement represents a logical IF statement. The logical IF will conditionally execute a statement following a logical expression. The logical expression following the IF and enclosed in parentheses is evaluated. If this logical expression is .TRUE., then the statement following the logical expression is executed. If it is .FALSE., then the statement is ignored. In either case, control then passes normally on to the next statement, unless

the statement following the logical expression was executed and caused a branch to another point in the program.

The statement following the logical expression may be any executable statement except a DO statement or another logical IF.

These are examples of logical IF statements:

a. IF (A.EQ.B.OR.C.EQ.D) G=G+1

If A equals B or C equals D (or both), then G will be incremented by 1. Otherwise, G will remain unchanged. In any event, control will then pass on to the next statement.

b. IF (L1) GO TO 97

If L1 (which must be declared to be a LOGICAL variable) is .TRUE., then control will pass to statement number 97. If L1 is .FALSE., then control will pass on to the next statement.

c. IF (A.LE.97) IF (B) 12,12,13

If A is less than or equal to 97, then the arithmetic IF will be executed, and control will pass to statement number 12 or 13, depending on the value of B. If A is greater than 97, control will pass on to the next statement.

• PAUSE STATEMENT

The executable PAUSE statement is provided to allow an executing program to be suspended indefinitely.

The proper format of the PAUSE statement is:

PAUSE or PAUSE p
where p is a one- to eight-digit unsigned integer constant or a proper or Hollerith string containing up to 17 characters.

The execution of the PAUSE statement causes the unconditional suspension of the program being executed pending operator action. The statement may appear at any point within any program unit except a BLOCK DATA subprogram. In addition to suspending the program, the execution of this statement causes the optional integer or string following the PAUSE to be displayed at the operator's console.

The program may be resumed at the first executable statement following the PAUSE statement via a system input message from the operator's console of the form:

mOK

where m is the mix number of the job. Alternately, the job may be discontinued at this point via a system input message of the form:

mDS

where m is the mix number of the job. The following are valid examples of the PAUSE statement:

```
PAUSE
PAUSE 2
PAUSE 'BEFORE PUNCHING' (for EBCDIC-coded input)
PAUSE "BEFORE PUNCHING" (for BCL- or BCD-coded input)
```

• RETURN STATEMENT

The executable RETURN statement is a control statement provided to specify the manner in which the calling program unit is to be reentered following the execution of a subprogram.

The proper forms of the RETURN statement are:

1. RETURN 2. RETURN a
where a is an arithmetic expression.

A SUBROUTINE or FUNCTION subprogram may cause termination of program execution by the execution of a STOP statement or return of control to the calling program unit by execution of a RETURN statement. The point in the calling program at which execution resumes is determined by the form of the RETURN employed.

A RETURN in a main program is interpeted as a STOP.

Standard Return

The execution of a RETURN statement of the first type causes a standard return to the calling program unit. For a subroutine, this means the next executable statement following the CALL statement invoking the subroutine will be the next statement executed. For a function, standard processing of the function reference will be performed and the calling program unit will continue executing in the standard fashion.

If the END statement of the subprogram is encountered before a RETURN statement is executed, a standard return will be performed. If no RETURN statement

is found in a subroutine or function during compilation, a warning message will be generated.

Nonstandard Return

Form 2 of the RETURN statement allows control to be returned to a specified labeled executable statement in the calling program unit. A nonstandard return from a function will not return a value for the function reference.

A nonstandard return is indicated in a subprogram by a RETURN followed by an arithmetic expression. If the value of this expression is not an integer, the value is rounded to the nearest integer. This final value, n, is used to select the n-th asterisk (*) in the dummy argument list of the SUBROUTINE or FUNCTION statement. If n is greater than the number of * items in this list or if n is less than or equal to zero, execution is terminated. If the expression is an integer constant which is zero, negative, or larger than the number of dummy asterisks, a syntax error results.

The statement label corresponding to the selected asterisk and found in the actual argument list preceded by an & will be used to identify the statement to which control is to be returned. The actual argument list of the CALL statement or function reference must contain such a statement label in each position where the dummy argument list of the SUBROUTINE or FUNCTION statement contains an asterisk. (See chapter 15.)

Example of Nonstandard Return

The following program excerpt illustrates a calling program unit able to handle nonstandard returns from the called subprogram.

```
      CALL SUBA(A1,&33,A2,&20,II)
15    Z=A1 + II
20    Z=Z + A2
33    Z=Z + A1
```

The called subroutine could be the following:

```
      SUBROUTINE SUBA (A,*,B,*,/J/)
      IF (A) 5,6,7
5     RETURN 1
6     J=J+B
      RETURN A+2
7     RETURN
      END
```

To consider this example, as a first case, suppose that when the IF statement in the subroutine is reached A is positive. Then there will be a branch to

statement number 7, which is a normal return. The subroutine will then return to the calling program unit at the statement following the CALL, statement number 15.

Supposing that A is less than zero, then when the IF statement in the subroutine is reached, there will be a branch to statement number 5. This is a nonstandard return to the first label in the actual argument list, since RETURN 1 selects the first asterisk in the dummy argument list. Then the return in the calling program unit will be to statement number 33.

If A is equal to zero, when the IF statement in subroutine SUBA is executed there will be a branch to statement number 6. As A = 0, when the arithmetic expression is evaluated it will read RETURN 2. This will cause a return to statement number 20 in the calling program unit.

Had a RETURN 3 or RETURN 0 been attempted in SUBA, execution would have been terminated.

A CALL statement that results in a nonstandard RETURN may be most easily regarded as a CALL followed by a computed GO TO. For example, the statement:

```
CALL SUBA(A1,&33,A2,&20,II)
```

may be treated as an effectively equivalent replacement for:

```
CALL SUBA(A1,A2,II,JUMP)
GO TO (33,20),JUMP
```

where JUMP is assigned a value of 1 or 2 (or some other value if return is to be standard) by the subroutine.

• STOP STATEMENT

The executable STOP statement is provided to allow the termination of an executing program.

The proper format of the STOP statement is:

STOP or STOP s
where s is a one- to six-digit unsigned integer constant.

The execution of the STOP statement causes the unconditional termination of the execution of the program being executed. This statement may appear at any point within any program unit except a BLOCK DATA subprogram.

The execution of a STOP statement or a CALL EXIT statement (for a program with no subroutine named EXIT) is the generally accepted manner in which a program may reach error-free termination. The optional integer following the STOP is ignored.

The following are valid examples of the STOP statement:

```
STOP
STOP 99
```

- **ZIP WITH STATEMENT**

The executable ZIP WITH statement is an auxiliary control statement and is discussed in chapter 16.

This statement allows system control information to be passed directly to the MCP from an executing program.

CHAPTER 9

ASSIGNMENT STATEMENTS

The executable assignment statements allow an arithmetic, logical, or label value to be assigned to a program variable.

The proper forms of the assignment statement are:

1. v=e 2. ASSIGN n TO k
where v and e are either a non-logical variable and an arithmetic expression or a logical variable and a logical expression, respectively; n is a statement label, and k is a simple single precision variable.

ARITHMETIC ASSIGNMENT STATEMENT

An arithmetic assignment statement is represented by form 1 of the assignment statement, with v being a non-logical variable and e being an arithmetic expression.

When such a statement is executed, the arithmetic expression is evaluated and the value obtained is placed into the storage word or word pair allocated to the variable, v.

In general, the variable and the arithmetic expression need not be of the same type. If the types are different, the expression is first evaluated and automatic conversion is subsequently performed on the value obtained to the type of the variable to be assigned the value. This automatic conversion proceeds according to the rules indicated in the following table.

The following notation is employed in the conversion table:

- n - indicates no conversion is performed;
- fix - indicates a conversion to integer with truncation;
- fl - indicates conversion from integer internal form to real internal form (which generally requires no action);
- real - indicates conversion by taking the "real" part of a complex item;

ASSIGNMENT STATEMENTS

rnd - indicates conversion from double precision to real with rounding;

ext - indicates conversion from real to double precision, with the second word set to zero;

i=0 - indicates that an "imaginary" part of zero is supplied.

	TYPE OF EXPRESSION			
TYPE OF v	INTEGER	REAL	DOUBLE PRECISION	COMPLEX
INTEGER	n	fix	rnd, fix	real, fix
REAL	fl	n	rnd	real
DOUBLE PRECISION	fl, ext	ext	n	real, ext
COMPLEX	fl, i=0	i=0	rnd, i=0	n

The internal storage formats of the various data types are discussed in chapter 11.

If a real value is to be converted to integer and the value lies outside the allowable range for integer constants, execution will be terminated.

As the system hardware automatically handles mixed-mode arithmetic, it is unimportant whether a value is stored in integer or floating-point format. The preceding table represents the situation when the parameter compiler option OPT is assigned the value -1. (See chapter 20.) When this option has the default value of 0, program values are stored in integer format when possible as a code optimization feature. Thus, relatively small floating-point values with a zero fraction part are stored as integers.

Examples:

```
G(IROW+2,JCOL)=IROW-N
A=I+12+B/3.6
C=6.2+(1,2.3)
```

LOGICAL ASSIGNMENT STATEMENT

A logical assignment statement is represented by form 1 of the assignment statement, with v being a LOGICAL variable and e being a logical expression.

When such a statement is executed, the logical expression is evaluated and the logical value obtained is placed into the storage word allocated to the variable, v.

Examples:

```
L(2,4)=L(1,1).AND.L(1,2)
D6=G.GT.H.OR.B.IS.C
L=.TRUE.
```

GO TO ASSIGNMENT STATEMENT

A GO TO assignment statement (or ASSIGN statement) is represented by form 2 of the assignment statement. This statement assigns the label of an executable statement to a simple integer variable in order that the variable may be used in an assigned GO TO statement. (See chapter 8.) The ASSIGN statement is the only statement that may give a variable the value of a label.

The following is an example of the use of the ASSIGN statement.

```
ASSIGN 999 TO LABEL
GO TO LABEL, (1,2,999)
```

When the GO TO statement is executed, control will be transferred to the executable statement bearing the label 999 in that program unit.

Until it has been reassigned through an arithmetic assignment statement, the referenced variable should not be used for purposes other than ASSIGN and GO TO statements since unpredictable results could occur.

CHAPTER 10

DATA STATEMENTS

The non-executable DATA statement is provided to allow compile-time initialization of program variables.

The proper form of the DATA statement is:

DATA k/d/ ...
where k is a list of variables to be initialized, d is an initial value list (see text), and the ellipsis (...) indicates as many repetitions of the form k/d/ as desired, with each repetition preceded by a comma.

STATEMENT USE

A DATA statement may appear at any point before the END statement in a program unit with the following exception. A DATA statement may not precede any specification statement affecting any variable initialized by the DATA statement.

The DATA statement is not an executable statement and is processed at compilation time. Variables appearing in a DATA statement in a subprogram will retain their values between references to that subprogram. A DATA statement appearing in a subroutine or function may not initialize elements of COMMON storage. Elements of a COMMON block may appear in DATA statements only in a BLOCK DATA subprogram or in a main program.

VARIABLE LISTS

The list of variables to be initialized in a DATA statement is constructed in the same manner as an I/O list with the following exceptions. (See chapter 12.)

a. Each element of the list should be initialized in all of the DATA statements of a program unit.
b. The initial parameter, terminal parameter, and the incrementation parameter in a DO-implied list must be unsigned integer constants or control parameters of another DO loop.

c. The control variable in a DO-implied list must appear only as a subscript in that list, and only array elements employing the control variable and constant integer subscripts may appear in such lists.

d. No arithmetic expressions may appear in the variable list.

The following is an example of a variable list:

```
K, M, (A(I,2), B(I),I=1,10),X
```

THE INITIAL VALUE LIST

The list of values to which the elements of the variable list are to be initialized consists of a list of constants and strings separated by commas.

The constants may optionally be preceded by a repeat count of the form n*, where n is an unsigned non-zero integer constant. This repeat count indicates the number of times the immediately following constant is to be interpreted.

Constant values are assigned to elements of the variable list in the order in which they occur. For example, the DATA statement:

```
DATA A,B/2,3/, C,D/2*4/
```

initializes the variables A and B to the values 2 and 3, respectively, and initializes C and D to 4.

The strings initial value list may contain Hollerith or proper strings up to 264 EBCDIC characters long. (A syntax error message is given during compilation if a longer string is encountered.) A "primary" string is defined to be a string consisting of six or fewer characters that initializes a single precision variable, or a string consisting of twelve or fewer characters that initializes a double precision variable. Similarly, a "long" string is defined to be a string consisting of more than six characters that initializes a single precision variable, or a string of more than twelve characters that initializes a double precision variable.

A primary string is stored left-justified within the variable with blank fill to the right if necessary. For example, if D is a DOUBLE PRECISION variable, the DATA statement:

```
DATA D/'bABCDEFG'/
```

(where b represents a blank) initializes D to the value: bABCDEFGbbbb.

Any long string initializing a simple variable or array element or preceded by a repeat count greater than 1 is truncated to a primary string, with the leftmost characters of the string retained.

A long string initializing an array name is transferred to the array, beginning with the indicated element, until either the end of the array or the end of the string is encountered. Character assignments are made to the array elements in the order in which these elements are stored internally. (See chapter 11.) If the string terminates before the entire array is initialized, the last initialized array element will be filled on the right with blanks, if necessary. Array elements beyond the last one initialized are set to zero.

For example, this program excerpt:

```
REAL A(6)
DOUBLE PRECISION D(2)
DATA D,A/'ABCDEFGHIJKLMNOPQRSTUVWXYZ',1,2,3,7HABCDEFG/
```

will cause these variables to be initialized to the indicated values:

```
D(1)='ABCDEFGHIJKL'
D(2)='MNOPQRSTUVWX'
A(1)=1
A(2)=2
A(3)=3
A(4)='ABCDEF'
A(5)='Gbbbbb'
A(6)=0
```

Double Precision Octal and Hexadecimal Constants

A double precision variable may be initialized by an octal constant containing 32 or fewer octal digits or by a hexadecimal constant containing 24 or fewer hexadecimal digits. Double precision items only may be initialized by these extensions to the octal and hexadecimal constants. The specified constant value will be stored right-justified within the two words of the variable thus initialized.

When 17 to 32 octal or 13 to 24 hexadecimal digits are specified in the constant, the constant is treated as two immediately adjacent items, one for each word of the double precision variable. The digits beyond those necessary to specify the contents of one word are stored right-justified in the second word.

For example, if D is a double precision variable, the DATA statement:

```
DATA D/Z123456789ABCDEF/
```

places the hexadecimal value 123456789ABC000000000DEF into D.

ASSIGNMENT CONSIDERATIONS

If a simple variable which is not in COMMON storage or EQUIVALENCEd is initialized more than once, a warning will be given during compilation. The variable will be initialized to the last value encountered. If such a variable or array is in COMMON storage or is EQUIVALENCEd and is initialized more than once, no warning will be issued during compilation but the value of the item will be undefined.

Conversion During Assignment

The following table indicates the type of conversion to be performed on a constant appearing in an initial value list when it is assigned as the initial value of a variable.

The following notation is used in this table:

n - no conversion (except as previously noted for strings);

fix - truncated to integer if magnitude does not exceed maximum integer size, integer overflow error given otherwise;

real - take the first word of the double precision value if the exponent field of the second word is zero, syntax error given otherwise;

ext - extend the single precision value to a double precision value with the second word zero;

i - invalid combination resulting in syntax error ("ILLEGAL MIXED TYPES").

	VARIABLE TYPE				
CONSTANT TYPE	INTEGER	REAL	DOUBLE PRECISION	COMPLEX	LOGICAL
Integer	n	n	ext	ext	n
Real	fix	n	ext	ext	n
Double Precision	real	real	ok	i	i
Complex	i	i	i	ok	i
Logical	i	i	i	i	ok
String	n	n	n	i	n
Single Precision Octal or Hexadecimal	n	n	ext	i	i
Double Precision Octal or Hexadecimal	i	i	n	i	i

CHAPTER 11

STORAGE ALLOCATION

Each B 6700/B 7700 FORTRAN data item is allocated one or more units of storage depending upon the type of value(s) the item represents. The primary unit of storage involved is the data word, 48 bits of which are accessible to the user.

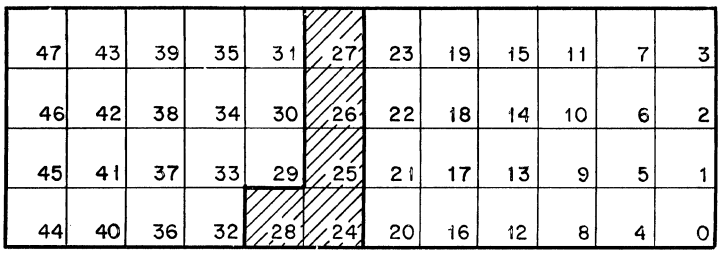
The following FORTRAN features require storage configurations:

a. Simple Variables
b. Arrays
c. EQUIVALENCEd Data Items
d. Elements of COMMON Storage

The method of storage allocation corresponding to each of these features is discussed in this chapter.

NOTE

The notation, [m:n], is used in this section to describe data word fields. The 48 accessible bits of a data word are considered to be numbered, with the left-most bit being bit 47 and the right-most bit being bit 0. In the notation used here, m denotes the number of the left-most bit of the field being described, and n denotes the number of bits in the field. For example, the word field shown here (bits 28 through 24) would be described by [28:5]:



STORAGE ALLOCATION

NOTE (Cont)

Hexadecimal constants are used extensively in the present chapter to indicate word contents. Such constants are particularly suited to describing the value of a data word as each digit in a hex constant indicates the contents of a 4-bit field. Such fields can be visualized as columns in the preceding picture of a data word.

SIMPLE VARIABLE

Each simple variable requires one or two words of storage depending upon the type of the variable. Each of these simple variable types corresponds to a special internal storage configuration:

a. Integer Variables
b. Real Variables
c. Double Precision Variables
d. Complex Variables
e. Logical Variables

The internal handling of variables of these five types is discussed here.

Integer Variables

An integer variable requires <u>one</u> word of storage. The data word corresponding to an integer variable is partitioned as follows:

FIELD	CONTENTS
[47:1]	Unused
[46:1]	Sign Bit
[45:7]	Contains all 0's
[38:39]	Magnitude of Data Item

Integer values are represented internally in signed-magnitude notation. The sign of the value is denoted by bit 46 of the data word involved. This bit is 0 for positive values and 1 for negative values. The magnitude of the value is stored right-adjusted in bits 38 through 0 preceded by zeros.

For example, the internal representation of the integer 10 described using hexadecimal constant notation is:

Z00000000000A

The internal representation of -10 is:

Z40000000000A

Due to this internal format, the largest integer value that may be stored is Z007FFFFFFFF = 549755813887.

A larger hexadecimal value would place a non-zero value in the [45:7] field of the data word, violating the internal format requirement of an integer value.

Real Variables

A real variable requires one word of storage. The data word corresponding to a real variable is partitioned as follows:

FIELD	CONTENTS
[47:1]	Unused
[46:1]	Mantissa Sign Bit
[45:1]	Exponent Sign Bit
[44:6]	Magnitude of Data Item Exponent
[38:39]	Magnitude of Data Item Mantissa

Real (floating-point) values are represented internally in signed-magnitude mantissa-and-exponent notation. The sign of the mantissa is denoted by bit 46 while the sign of the exponent is denoted by bit 45 of the data word. A minus sign is denoted by a 1 in the appropriate bit. The magnitude of the exponent of the data item is contained in bits 44 through 39; hence, the exponent may not exceed a magnitude of (2**6) -1 (i.e., 63). The magnitude of the mantissa of the data item is contained in bits 38 through 0. This latter word field is identical to the magnitude field of a data word containing an integer value. Thus, an integer data word may be considered to be a real data word with an exponent field containing all zeros.

STORAGE ALLOCATION

The value represented by a real data word may be obtained by the formula:

(mantissa) * 8 ** (exponent)

The magnitude of the mantissa is usually stored left-normalized within its word field. Therefore, the internal representation of the real value 0.5 (or .5E0) in hexadecimal and octal constant notation is Z26C000000000 or 0115400000000000000, respectively. The exponent in this example is -13 and the mantissa is 4*8**12.

Double Precision Variables

A double precision variable requires two adjacent words of storage. The data words corresponding to a double precision variable are partitioned as follows:

FIELD	CONTENTS
(FIRST WORD)	
[47:1]	Unused
[46:1]	Mantissa Sign Bit
[45:1]	Exponent Sign Bit
[44:6]	Least Significant Part of Magnitude of Exponent
[38:39]	Most Significant Part of Magnitude of Mantissa
(SECOND WORD)	
[47:9]	Most Significant Part of Magnitude of Exponent
[38:39]	Least Significant Part of Magnitude of Mantissa

Double precision values are represented internally in signed-magnitude mantissa-and-exponent notation with a most significant part and a least significant part. The sign of the mantissa of the data item is contained in bit 46 of the first data word while the sign of the exponent is contained in bit 45 of the first word. A minus sign is denoted by a one in the appropriate bit. The magnitude of the exponent of the data item is contained in a total of 15 bits. The first

nine of these 15 bits (the most significant part of the magnitude of the exponent) are contained in bits 47 through 39 of the second data word. The remaining six of these 15 bits (the least significant part of the magnitude of the exponent) are contained in bits 44 through 39 of the first data word.

The magnitude of the mantissa of the data item is contained in a total of 78 bits which are divided in half with each half being placed into one of the two data words. The first 39 bits (the most significant part of the magnitude of the mantissa) are contained in bits 38 through 0 of the first data word. The remaining 39 bits (the least significant part of the magnitude of the mantissa) are contained in bits 38 through 0 of the second data word. The mantissa sign bit applies to both portions of the mantissa.

The value represented by a double precision data word pair may be obtained by the formula:

```
((most significant part of mantissa)
+(least significant part of mantissa)*8**(-13))
*8**((most significant part of exponent)
   +(least significant part of exponent)*2**9))
```

The magnitude of the mantissa usually is stored left-normalized within the total 78-bit field in which it is contained.

As an example, the internal two-word representation of the double precision value 1D0 in hexadecimal constant notation is:

```
Z261000000000
Z000000000000
```

The contents of the first of these two data words is identical to the contents of a data word representing the real value 1E0. When a value which does not exceed the limits of a real variable is assigned to a double precision data word pair, then that data word pair can be represented in a form that will contain all zeros in the second data word while the first data word will contain a representation of the value identical to that which would have been formed if the value had been assigned to a real variable.

Complex Variables

A complex variable requires <u>two</u> adjacent words of storage. The first of these two data words contains the "real" part of the variable, while the remaining data word contains the sign and magnitude of the "imaginary" part of the variable.

STORAGE ALLOCATION

These two data words conform to the specifications governing data words corresponding to integer and real constants. If the complex variable is assigned the value of a complex constant whose "real" and/or "imaginary" part is an integer constant, then the first and/or last data word corresponding to the complex constant will be structured as an integer data word. A real constant in the complex constant will similarly control the structure of the corresponding complex data word.

For example, the assignment statement:

```
C = (1,0.5E0)
```

places into the two data words corresponding to the previously-declared complex variable C these values:

```
Z000000000001
Z26C000000000
```

The first data word (i.e., the word containing the "real" part of C) contains a representation of the integer constant 1. The second data word (i.e., the word containing the magnitude of the "imaginary" part of C) contains a representation of the real constant 0.5E0. If the assignment statement were:

```
C = (1, 1)
```

then the contents of both data words would be identical as:

```
Z000000000001
Z000000000001
```

Logical Variables

A logical variable requires <u>one</u> word of storage. The data word corresponding to a logical variable is partitioned as follows:

FIELD	CONTENTS
[47:47]	Unused
[0:1]	Value Bit

The left-most 47 accessible bits of the data word containing the value of a logical variable are ignored, while the value represented by that variable is contained in bit 0 of the data word. This logical value is represented as follows:

When bit 0 is 1, the value of the variable is .TRUE. and
when bit 0 is 0, the value of the variable is .FALSE.

Usually, the internal representation of a logical value is identical to the representation of an integer 1 or an integer 0. However, since only the state of bit 0 is taken into consideration, the remaining 47 bits may assume any states. Logical operators reference all 48 bits. (See chapter 6.)

For example, a logical variable may be EQUIVALENCEd to any odd-valued (i.e., 1, 3, 5, etc.) integer variable and the value of the logical variable will be .TRUE.. When EQUIVALENCEd to any even-valued (i.e., 0, 2, 4, etc.) integer variable, a logical variable will have the value .FALSE..

ARRAYS

FORTRAN arrays are provided to allow the user to organize program storage locations into a convenient structure. Internally, an array is stored as a group of one or more contiguous data words. The correspondence between the program array elements and the group of internal storage words is discussed here.

A FORTRAN array of any number of declared dimensions is represented internally by a one-dimensional array (i.e., a vector) of storage locations. Each element of the array has storage requirements identical to that of a simple variable of the same type as the array. Thus, each element of a REAL array requires one word of storage while each element of a COMPLEX array requires two words of storage. The partitioning of each storage word is identical to that of the storage word(s) corresponding to a simple variable of the same type as the array element.

Each INTEGER, REAL, and LOGICAL array is allocated a series of internal data words exactly equal in number to the elements of the array. DOUBLE PRECISION and COMPLEX arrays are allocated twice as many internal data words as array elements. For one-dimensional arrays the internal data word or word pair corresponding to each array element occurs in the same position in the internal array as the element occurs in the program array.

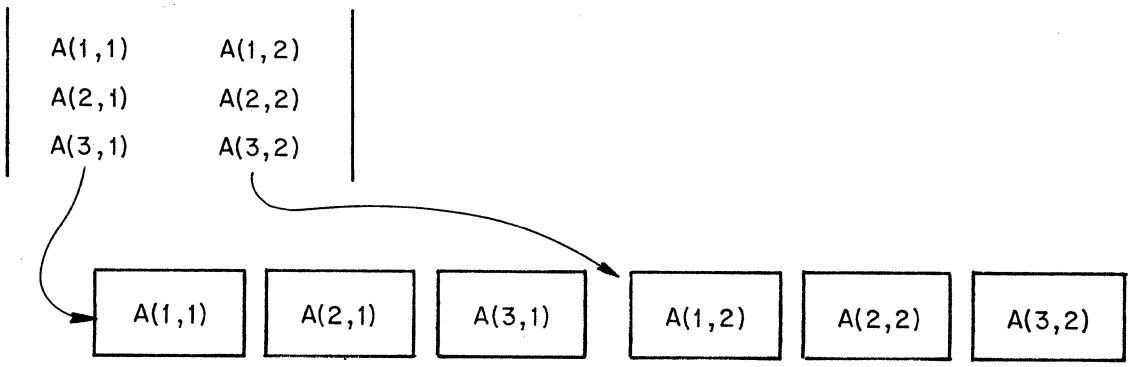
For example, if the arrays A1 and A2 are declared using these statements:

```
DIMENSION  A1(20)
COMPLEX    A2(20)
```

then the REAL array A1 will be allocated 20 words of storage (one word per element), and the COMPLEX array A2 will be allocated 40 words of storage (two words per element). In addition, the array element A1(2) will be assigned the second word of the internal array corresponding to A1, and A2(2) will be assigned the third and fourth words of the internal array corresponding to A2.

STORAGE ALLOCATION

Arrays of more than one dimension are stored by columns into one continuous internal vector of storage words. This storage process may be visualized as occurring in the manner displayed in the example provided here.



The word(s) corresponding to each element of the two-dimensional array in this example is located in the internal array in the order shown. Beginning with element A(1,1), each successive element in the array may be thought of as being taken in the order in which they occur, proceeding down each column of the matrix, from left to right, and stored in the internal array. For three-dimensional arrays, this process is repeated for each successive layer of the array (i.e., each value of the third subscript, beginning with 1).

Thus, for any n-dimensional array, the elements of this array correspond in a set order to successive storage locations in the one-dimensional internal array associated with the program array. The appropriate order is identical to that obtained when one lists all of the array elements by varying the first subscript most rapidly, the second subscript next most rapidly, and so forth. As an example, the elements of the five-dimensional array A(2,1,2,2,1) are stored in this order:

A(1,1,1,1,1)
A(2,1,1,1,1)
A(1,1,2,1,1)
A(2,1,2,1,1)
A(1,1,1,2,1)
A(2,1,1,2,1)
A(1,1,2,2,1)
A(2,1,2,2,1)

Element A(1,1,1,1,1) in this example corresponds to the first element of the internal array, element A(2,1,1,1,1) corresponds to the second element of the internal array, and so forth.

An array A having N dimensions may be declared via an array declarator of the type:

A(D1,D2,D3,...,DN)

The array A then contains elements of the form:

A(I1,I2,I3,...,IN)

The position of the storage unit (i.e., the word or word pair) assigned to this array element within the corresponding internal array may be found via the formula (called the array element successor function):

$$I = I1 + D1*(I2-1) + D1*D2*(I3-1) + \ldots + D1*D2*D3*\ldots*DN-1*(IN-1)$$

where I is the location index between 1 and (D1*D2*D3*...*DN), inclusive. Thus, the linear index from the first address of the internal array to the array element is one less than this value (times 2 for DOUBLE PRECISION and COMPLEX arrays).

An array having more than one dimension may be EQUIVALENCEd to a one-dimensional array and the elements of this latter array will correspond in order to the storage units assigned to the former array. This order is also the order in which array elements are considered when an array name appears without a subscript list within the list in an I/O statement or as the item to be initialized in a DATA statement.

A program array is stored internally in consecutive storage locations as long as the storage requirement of the array does not exceed 1023 words. Arrays exceeding this size will be segmented for overlay purposes unless the compiler option LONG is set. (See chapter 19.) Each array segment will contain a maximum of 256 data words and will be automatically overlayed and recalled as necessary during the execution of the program. In no case may a program array exceed 65,535 words in length (i.e., 256 array segments).

STORAGE ALLOCATION

Array segmentation may be suppressed through use of the LONG compiler control option. (See chapter 19.) When the LONG option is set, program arrays requiring more than 1023 words of storage will not be segmented; the entire array will be overlayed and recalled as necessary during program execution. The process of array segmentation minimizes the use of core storage during program execution, but execution time is spent during the retrieval of array segments from overlay memory as portions of the array are required by the program. The suppression of array segmentation minimizes the time spent in the retrieval of array segments from overlay memory and element addressing, but a considerable amount of core storage is required to maintain such an array in memory. Hence, the user should allow array segmentation to occur if core memory use is to be kept at a minimum, or suppress segmentation (by setting the LONG compiler option) if execution time is to be minimized.

EQUIVALENCED DATA ITEMS

The FORTRAN EQUIVALENCE specification statement allows the user to assign a number of program data items to a single unit of internal storage. Thus, more than one symbolic name (i.e., variable name or array element) may refer to one storage location.

The programming considerations involved with the use of the EQUIVALENCE statement are discussed here.

Single Storage Locations

The least complicated use of the EQUIVALENCE statement involves the assignment of data items requiring a single word of storage to a mutual storage location. As an example, assume that the following statements are the first statements of an executable program.

```
IMPLICIT INTEGER (A), REAL(B), LOGICAL(L)
DIMENSION AR(2)
EQUIVALENCE (A, AR(2), B2), (AL,L)
```

The displayed EQUIVALENCE statement causes the INTEGER variable A, the INTEGER array element AR(2), and the REAL variable B2 to all be assigned to one data word. The first element of AR is not affected by this specification statement. A change in the value of any one of the three EQUIVALENCEd items produces a simultaneous, equivalent change in the value of the other two items.

The EQUIVALENCE statement also causes the INTEGER variable AL and the LOGICAL variable L to be assigned to the same data word. As the variable L changes value, AL also changes value. For example, the assignment statement:

```
L = .TRUE.
```

places the integer value 1 into variable AL. Also, as the variable AL changes value, L also changes value accordingly. As AL assumes even integer values (e.g., 0, 2, -4, etc.), L assumes the value .FALSE., and as AL assumes odd integer values (e.g., 1, 3, -5, etc.), L assumes the value .TRUE..

Multiple Storage Locations

EQUIVALENCE statements may also involve data items requiring more than one word of storage. As an example, assume that the following statements are the first statements of a program.

```
DOUBLE PRECISION D
REAL A(2)
EQUIVALENCE (A, D, B)
```

The displayed EQUIVALENCE statement causes the REAL array A, the DOUBLE PRECISION variable D, and the REAL variable B to be assigned to identical data words. As A and D both require two data words, the first and second elements of A become equivalent to the first and second words, respectively, of the storage unit assigned to D. The variable B, requiring only one data word, will be assigned to the same location as A(1) and the first word of D.

The equivalencing of a two-element array to a DOUBLE PRECISION variable is particularly useful in one respect. It allows a string of length 12 to be assigned to the DOUBLE PRECISION variable by assigning strings of length 6 to each element of the array as in the following example:

```
      DOUBLE PRECISION STRING
      DIMENSION PART(2)
      EQUIVALENCE(STRING, PART)
      PART(1) = 'ABCDEF'
      PART(2) = 'GHIJKL'
      WRITE(6,6) STRING
6     FORMAT(X,A12)
      END
```

This sample program produces the following output:

```
 ABCDEFGHIJKL
```

STORAGE ALLOCATION

Array Handling

The EQUIVALENCE statement may be used to assign a single group of contiguous storage locations to a number of program arrays. The present discussion illustrates the effect of the appearance within an EQUIVALENCE list of each of these three types of possible array references:

a. An array name.
b. An array element with the same number of subscripts as contained in the declarator declaring the array.
c. An array element with one subscript from an array of more than one dimension.

Assume that the following statements are the first statements of a program unit.

```
REAL A(4), B(10, 10), C(100), D(50), E(3, 3), F(50)
DOUBLE PRECISION $DE(2) / 2* 1D0/
EQUIVALENCE (A,$DE), (B(1,1),C(1)),(D,F(26),E(2))
```

The first list in this EQUIVALENCE statement (i.e., (A,$DE)) causes the REAL array A and the DOUBLE PRECISION array $DE to share four storage words. The first two elements of A (i.e., A(1) and A(2)) become equivalent to the two words of the first element of $DE (i.e., $DE(1)), and the last two elements of A (i.e., A(3) and A(4)) become equivalent to the two words of the last element of $DE (i.e., $DE(2)). The appearance of array names only in an EQUIVALENCE list causes equivalencing to begin with the first element of each array. The initial value list in the type statement declaring $DE causes the double precision value 1D0 to be placed in $DE(1) and $DE(2) and the word pairs A(1), A(2) and A(3), A(4).

The second list in the sample EQUIVALENCE statement (i.e., (B(1,1), C(1))), causes the REAL arrays B and C to share 100 storage words. Each element in the 100-element array C (beginning with C(1)), is assigned to the same storage location as a unique element of B. The elements of the two-dimensional array are stored internally in the manner described in the discussion of "ARRAYS" in this chapter. The internal storage locations assigned to C occur in the same order as the elements of that program array. Hence, each C(I) is

equivalenced to the Ith internal element of B and it follows that if the following two PRINT statements occur in the same program unit, identical output will be produced:

```
PRINT 1,(C(I),I=1,100)
PRINT 1,B
```

The final list in the sample EQUIVALENCE statement (i.e., (D,F(26),E(2))) indicates that the elements of the arrays D, F, and E are to be equivalenced in such a manner that D(1), F(26), and the second internal element of the two-dimensional array E are to be assigned identical internal locations. Therefore, the last 25 elements of F (i.e., F(26) through F(50) become equivalent to the first 25 elements of D (i.e., D(1) through D(25)). Since E is stored internally in the manner described in the discussion of ARRAYS, equivalencing is handled in the manner illustrated below. Each of the following lines denotes a single storage location, and the array element(s) on a line is assigned to the corresponding location.

	F(1) through F(25)	E(1,1)
D(1)	F(26)	E(2,1)
D(2)	F(27)	E(3,1)
D(3)	F(28)	E(1,2)
D(4)	F(29)	E(2,2)
D(5)	F(30)	E(3,2)
D(6)	F(31)	E(1,3)
D(7)	F(32)	E(2,3)
D(8)	F(33)	E(3,3)
D(9) through D(25)	F(34) through F(50)	
D(26) through D(50)		

Therefore, the following DATA statement would initialize the elements E(1,2), E(2,2), F(28), F(29), D(3), and D(4) with the value 6:

```
DATA (E(J,2), J=1,2) / 2*6/
```

STORAGE ALLOCATION

ELEMENTS OF COMMON STORAGE

The FORTRAN COMMON specification statement allows program values to be communicated between program units without employing entries in SUBROUTINE and FUNCTION statement argument lists and permits these data items to be referenced by the same or different symbolic names in each program unit. This statement also allows data initialization via the BLOCK DATA program unit. (See chapter 15.)

The storage handling of elements in COMMON blocks referenced in subprograms is performed in a different manner from dummy arguments in such subprograms. Each element of a COMMON block is allocated storage in COMMON storage once for an executable program. Each program unit may reference a COMMON block (and hence each location in the block) via an appropriate COMMON statement. The contents of the locations thus referenced may be changed in the same manner as the contents of any location local to the program unit.

The size of each block of COMMON storage is either as large as the maximum specification indicated by a COMMON statement referencing the block name in any program unit or as large as the maximum length to which the block is extended by an EQUIVALENCE statement. These two concepts are discussed in the following paragraphs. In no event may a COMMON block exceed 65,535 data words in length.

Assume that the following statements are the initial statements of a program unit.

```
SUBROUTINE MSG
DOUBLE PRECISION D
LOGICAL FLAG(6)
COMMON WORD1,WORD2,D,FLAG,TEXT(20)
COUNT=1
```

The COMMON statement among the preceding statements will be assumed to provide the largest description of the size of the unlabeled COMMON block of any COMMON statement appearing in the executable program of which the program unit MSG is a part. The total size of this COMMON block is then 30 words. These words are recognized in the MSG subprogram as the words assigned to the REAL variables WORD1 and WORD2, the word pair assigned to the DOUBLE PRECISION variable D (which begins as required at an odd-numbered word within the block), the six words assigned to the LOGICAL array FLAG, and the twenty words assigned to the REAL array TEXT. These data words are contained at relative locations within the COMMON block in the order listed.

The unlabeled COMMON block just described could be referenced, for example, by a COMMON statement within another program unit as indicated here:

```
      SUBROUTINE DUMP
      COMMON T(10)
      PRINT 1,T
    1 FORMAT(X,Z12)
      RETURN
      END
```

where T is a REAL array. The elements of this program array would therefore be assigned the data words contained in the COMMON block beginning with the initial word of the block and proceeding for ten words. Thus, WORD1 and WORD2 are equivalent to the array elements T(1) and T(2), respectively; D is equivalent to the elements T(3) and T(4); and FLAG(1) through FLAG(6) are equivalent to elements T(5) through T(10). The data words allocated to the TEXT array in the MSG subprogram are not accessed in the DUMP subprogram.

Additionally, the EQUIVALENCE statement may interact with the COMMON statement to expand the length of a COMMON storage block beyond the maximum size specified for the block by the COMMON statement. It is possible to equivalence the beginning of an item representing more than one storage location (such as an array) to an element of the COMMON block and result in the addition of storage locations at the end of the block. The following discussion will illustrate this point.

As an example, assume that the following statements form two units of an executable program.

```
      FUNCTION SUM(N)
      COMMON /GR1/IT(3,3)
      DO 1 I=1,3
      DO 1 J=1,3
    1 SUM=SUM+IT(I,J)
      SUM=SUM*N
      RETURN
      END
      LOGICAL FUNCTION TEST(L)
      LOGICAL X(6)
      COMMON /GR1/K(9)
      EQUIVALENCE (K(6),X)
      DO 1 I=1,9
    1 S=S+K(I)
      TEST=S.EQ.L.AND.X(1).AND.X(6)
    C ELSE TEST IS .FALSE.
      RETURN
      END
```

STORAGE ALLOCATION

The COMMON block referenced by these two sample program units is labeled GR1. The function SUM accesses the first nine locations of this block through the two-dimensional INTEGER array IT. The function TEST accesses the first nine locations of the block via the INTEGER array K. In addition, the following two locations of the GR1 block are referenced as the LOGICAL array elements X(5) and X(6) since the X array is equivalenced to the K array starting at the element K(6).

The elements of the K array occur in the same order as the contiguous storage locations assigned to the array IT but allow these locations to be referenced using only one subscript. Equivalencing portions of the X and K arrays allow various elements of K to be handled as both INTEGER and LOGICAL type items.

The COMMON block could not have been extended backwards by the EQUIVALENCE statement. The following combination of statements is invalid:

```
LOGICAL X(6)
COMMON/GR1/K(9)
EQUIVALENCE (X(3),K)
```

CHAPTER 12

INPUT/OUTPUT STATEMENTS

The FORTRAN constructs which handle communication of data between internal and external storage consist of the executable Input/Output (I/O) statements. The external storage units involved are the program files. This chapter discusses these program files and the manner in which they are accessed via the I/O statement constructs.

The B 6700/B 7700 FORTRAN I/O statements are discussed in detail in the following paragraphs. In the descriptions of the forms of these statements, the following notation is employed:

n - denotes a file designator.
m - denotes an I/O list.
a - denotes a result clause list.

The file designator indicates the program file referenced, the I/O list specifies the data items involved in the action initiated by the I/O statement, and the result clause list allows for error handling. These three constructs are discussed first.

PROGRAM FILES AND FILE DESIGNATORS

Program files are units of external storage containing sequential records of program data. The records of these files are contained on peripheral hardware devices. These files may be input or output card files, input or output disk, disk pack or magnetic tape files, output line printer listings, or files assigned to remote terminals. Other devices are allowed via a FILE card.

A physical device may have more than one program file associated with it. The records contained in these files are each an integer number of data words in length and exist as a totally ordered set. The N records of a file are logically numbered in order in integer increments from 1 through N; record 1 immediately follows the initial point of the file, while record N immediately precedes the endfile mark terminating the file.

No file designator may refer to more than one program file at a time. A file designator on this system is an unsigned one- or two-digit non-zero integer constant which allows program I/O statements to reference the associated file.

INPUT/OUTPUT STATEMENTS

An I/O statement may employ an arithmetic expression to designate a file, in which case the expression is truncated to obtain an integer file designator. The following statements illustrate the use of file designators in I/O statements.

```
READ(5,1,END=99)A        (file designator = 5)
WRITE(6,33)              (file designator = 6)
FIND (99=7)              (file designator = 99)
ENDFILE 4.6              (file designator = 4)
```

Each program file is recognized by the system via two names: the internal name and the external name or TITLE (i.e., the value of the TITLE attribute of the file). (See Input/Output Subsystem Information Manual, Form No. 5000185.) The internal file name is constructed of the identifier "FILE" with the file designator appended to it. Thus, the form of an internal file name is:

FILEn

where n is the corresponding one- or two-digit file designator. The internal file names corresponding to the file designators 6 and 99 are FILE6 and FILE99, respectively. The internal name of a file may be used on a FILE system control card, or an INCLUDE compiler control card, or may be used by a bound-in, separately compiled non-FORTRAN program unit which references that file.

The TITLE of a file consists of one to fourteen identifiers separated by slashes (/); each identifier is one to seventeen characters long. This file TITLE may be specified on a FILE system control card, on a file declaration card (as discussed in appendix B), or on an INCLUDE compiler control card. (See chapter 19.) The form of the file TITLE does not necessarily bear a direct relation to the file designator as does the internal name. When the TITLE of a program file is unspecified, a default TITLE is assigned to the file which is identical to the internal file name. For example, the default TITLE assigned to the file referenced by the designator 4 is the internal file name which is FILE4. The following are examples of valid file TITLEs:

```
A/B
FILE7
PRINTER
DATA-FILE/ONE/FINAL
```

File names are discussed in appendix B and in chapter 19 (INCLUDE compiler option).

Care must be taken when employing file designators. A syntax error will be given if a source program is compiled in which both of these two conditions exist:

a. the only I/O statements which appear employ file designators that are arithmetic expressions containing more than just a real or integer constant, and

b. no file declaration (FILE) cards appear at the start of the program unit.

The compiler must be able to assign file values at compilation time, and it is unable to do so if the value of the file designator(s) employed is not readily available to it.

If such file designators are available at compilation time and if at any point in the program execution an I/O statement references any file which does not correspond to the designators found during the compilation, then the program execution will be terminated by the system and the message:

UNDEFINED VARIABLE FILE

will be displayed at the operator's console.

Each file designator has a default type of file associated with it. File designators 5, 6, and 7 are assigned to card reader, line printer, and card punch files, respectively, by default. Other designators are assigned to tape files by default.

The type of file with which a file designator is associated may be specified via a file declaration card preceding the input source deck. These declaration cards are described in appendix B and are provided to allow the user to assign values to attributes of program files. (File attributes are discussed in the Input/Output Subsystem Manual (5000185) and in the B 6700/B 7700 System Software Handbook (5000722.)

The FORTRAN compiler automatically assigns values to file attributes when a constant file designator is encountered in I/O statements in the source input. The effect of the appearance of such file designators is identical to the effect which would have been produced had a file declaration card been used assigning default values to the attributes of the file associated with the designator. Hence, the user need never employ file declaration cards if the default file specifications are satisfactory and consistent with the intended use of the file. Program files may be assigned to the following devices: line printer, card reader, card punch, disk, remote terminal, disk pack, and 7- and 9-track magnetic tape. Multi-file tape reels are discussed at the end of this chapter.

INPUT/OUTPUT STATEMENTS

Any BACKSPACE, FIND, PRINT, PUNCH, READ, or WRITE statement will open a file if it is not already open.

I/O LISTS

I/O lists are used as elements of various I/O statements (e.g., READ, WRITE) to denote the program storage locations referenced by the execution of these I/O statements. Each I/O list consists of one or more symbolic names denoting storage locations to or from which data is to be transferred when the statement is executed. Data is transferred in the order indicated by the ordering of the names in the list.

An I/O list may be a simple I/O list, a DO-implied I/O list, or two I/O lists separated by a comma (,). A simple I/O list consists of an array name, an arithmetic expression, or a logical expression. The following are examples of valid I/O lists:

```
A,B(3)
XARRAY, XARRAY (2,4)
1+COS(N) , X(1+N), G
RESULT
A .OR. B
```

A DO-implied I/O list is an I/O list immediately followed by a comma and a DO-implied specification, all of which is enclosed within a set of parentheses. A DO-implied specification is constructed in the same manner as the DO-control portion of a DO statement. (See chapter 8, CONTROL STATEMENTS.) Thus, the proper format of a DO-implied I/O list is:

```
(m, v=p,q,i)
```

or

```
(m, v=p,q)
```

where m is an I/O list, v is the single precision control variable, and p, q, and i are arithmetic expressions which are the initial parameter, terminal parameter, and incrementation parameter, respectively. The control variable need not appear in the I/O list part of this construct. The following are examples of valid DO-implied I/O lists:

```
(A(I) , B , I , C(I,J) , I = 1,10,2)
(IPLOT , COUNT = 1,60)
(X, (Y(I,J), I=1,10), Z, J=1,2)
```

An I/O list may consist of any combination of simple and DO-implied I/O lists separated by commas. Such a list may contain grouping parentheses. The following are examples of valid I/O lists:

```
A, (I,GX(I), B, I=3,5),10,X
XARRAY(3), XARRAY,(XARRAY(EL), EL = 1,20)
RESULT
(A(I), I=1,20) , (J(I), I=1,5)
(((Z(I,J,K),K=1,10),J=1,10),I=1,10),7,G
```

Transfer of data to and from storage locations occurs in the order these locations are named in the I/O list, from left to right. Items named in a DO-implied I/O list are referenced repeatedly until the implied DO is satisfied. The appearance in an I/O list of an array element specifies only that array element, while the appearance of an array name specifies every element declared for the array in that program unit. These array elements are transferred in the order in which they are stored in the internal array assigned to the program array. (See discussion of array storage in chapter 11.)

An arithmetic expression may appear in an I/O list in which case the value of the expression is used to provide FORMAT information (to replace an * or V FORMAT specification) or to provide a value for output. If the expression invokes a function, that function must not cause execution of an I/O statement which employs the same program file as the file employed by the statement using the I/O list which contains the expression. Thus, if this I/O statement is executed:

```
WRITE (6,1)  A, F (20,X)
```

then F must not be a function which contains an I/O statement employing FILE6 since it is the file employed by this WRITE statement.

RESULT CLAUSE LISTS

Lists of I/O result clauses may be employed as elements of READ and WRITE statements containing file designators. Such lists are provided as error handling aids and consist of one or more I/O result clauses separated by commas.

The presence of a result clause list within a READ or WRITE statement may prevent termination of the program in the event of an error incurred during data transfer.

An I/O result clause is defined as either an action specifier or result descriptor assignment.

• Action Specifier

An action specifier is defined as one of the following constructs:

DATA = s END = s ERR = s
where s is an unsigned integer constant corresponding to the label on an executable statement appearing in the same program unit as the I/O statement.

If the DATA action specifier appears as a result clause in a formatted READ statement, then control will pass automatically to the statement bearing the label indicated by s when the data read does not agree with the requirements specified by the FORMAT statement. If the DATA clause appears in an unformatted READ or WRITE statement, then control will pass automatically to the specified statement if the number of data words indicated by the I/O list exceeds the size of the accessed record. If these conditions occur and the DATA clause is missing, then the system will terminate the program execution.

If the END action specifier appears as a result clause in a READ or WRITE statement, then control will automatically pass to statement s when an end-of-file condition occurs. Such end-of-file conditions occur when the following actions are attempted by a READ or WRITE statement.

STATEMENT	TYPE OF FILE	ACTION CAUSING END-OF-FILE CONDITION
READ	CARD	Attempting to read a card with an invalid character in column 1.
READ	TAPE	Attempting to read an End-of-File record.
READ	DISK	Attempting to read a record beyond the last record of a file.
WRITE	DISK	Attempting to write beyond the end of the designated area of the disk file.
WRITE	LINE PRINTER	Attempting to write a record at end-of-page. The PAGESIZE file attribute must be assigned an appropriate value on a FILE system control card or file declaration card (see appendix B). Also, see B 6700/B 7700 System Software Handbook (5000722).

If the ERR action specifier appears as a result clause in a READ or WRITE statement, then control will automatically pass to statement s when a parity error occurs during the data transfer. The block in which the parity error occurs is available for processing. The ERR action specifier will avoid termination of the program at this point, and the next block may be processed if no parity errors occur in it.

Examples

The following are valid examples of action specifiers:

```
DATA = 12
END = 20006
ERR = 99
```

- **Result Descriptor Assignments**

A result descriptor assignment is defined as the following construct:

RESULT = v
where v is a variable name or array element.

A FORTRAN program may obtain information regarding the result of an input or output operation via the result descriptor word returned by the system I/O routines. The result descriptor assignment clause allows a variable or array element to be named as the recipient of this result descriptor. The result word returned contains the following information. (The notation used here to describe data word fields is discussed in chapter 11.)

FIELD	CONTENTS
[0:1]	A 1 indicates an error has occurred during the I/O operation and that the field [16:16] is non-zero. A 0 indicates a successful I/O operation.
[4:1]	A 1 indicates a data error has occurred.
[7:1]	A 1 indicates a parity error has occurred.
[9:1]	A 1 indicates an EOF or end-of-page occurred.
[15:1]	A 1 indicates a time-out has occurred.
[16:1]	A 1 indicates a file security violation occurred.
[24:8]	The number of characters in the last word of the record processed if unblocked.
[47:20]	Actual data size of the logical record in INTMODE units or words as controlled by the UNITS file attribute.

If the variable or array element specified in the RESULT clause is of LOGICAL type, then that variable will be TRUE if an I/O error has occurred.

The following is a valid example of a result descriptor assignment:

```
RESULT = A(1,2)
```

Examples of Result Clause Lists

A result clause list may consist of a minimum of one I/O result clause as illustrated in this list example:

```
RESULT = RESULT(2)
```

The result clause list just presented specifies that the result word obtained from the system I/O routine invoked by the statement bearing this list is to be placed in the second element of the one-dimensional array named RESULT. The program will not be terminated if an error occurs at this point.

If this result clause list appears as an element of a READ or WRITE statement:

```
END = 99, ERR = 601
```

then control will immediately transfer to statement 99 or 601 if an end-of-file or irrecoverable parity error condition, respectively, arises during the execution of the I/O statement.

I/O STATEMENT SYNTAX

FORTRAN I/O statements may be grouped in the following manner:

a. Input statement (i.e., READ statement),
b. Output statements (i.e., WRITE, PRINT, and PUNCH statements), and
c. Auxiliary I/O statements (i.e., BACKSPACE, CHANGE, CLOSE, ENDFILE, FIND, INQUIRE, LOCK, OPEN, PURGE, and REWIND statements).

The following paragraphs discuss the syntax of these statements in the order just listed.

• READ STATEMENT

The executable READ statement allows data to be input from peripheral storage into the internal storage locations indicated by the I/O list or format portion of the statement. The transferred data may be converted during the transfer process as indicated by a specified FORMAT statement or format specification item, or read without editing if no format items are specified. Provision may also be made to handle errors incurred during the read.

The proper forms of the READ statement are:

```
 1.   READ f,m
 2.   READ (n,f)m
 3.   READ (n=r,f)m
 4.   READ (n,f,c)m
 5.   READ (n=r,f,c)m
 6.   READ f
 7.   READ (n,f)
 8.   READ (n=r,f)
 9.   READ (n,f,c)
10.   READ (n=r,f,c)
11.   READ (n)m
12.   READ (n=r)m
13.   READ (n,c)m
14.   READ (n=r,c)m
15.   READ (n)
16.   READ (n=r)
17.   READ (n,c)
18.   READ (n=r,c)
19.   READ (a,f)m
20.   READ (a,f,c)m
21.   READ (a,f)
22.   READ (a,f,c)
23.   READ (a)m
24.   READ (a,c)m
```

where n and r are arithmetic expressions representing a file designator and record number, respectively; f is either the label of a FORMAT statement, an array name denoting an array containing format specification, a NAMELIST name (in forms 6-10), or a slash (/) for free-format input (except forms 6-10); c is a result clause list; m is an I/O list; and a is an array identifier designating an internal data transfer. (See below.)

Files Referenced

Execution of any READ statement, forms 1 through 18, causes data to be read from one or more records of a program file. The file referenced is either indicated implicitly through use of READ statement forms 1 or 6, or explicitly through use of any of the other forms via the file designator, n. READ statement forms 1 and 6 will reference the same file as a READ statement using a file designator having the value (when truncated) of 5 unless a file declaration card with the designator "READER" or a READER compiler option card applies to the present program unit. (See appendix B.) If such a file declaration card appears, READ statement forms 1 and 6 will reference the file declared on this card. The default kinds of files referenced by various file designators are discussed in appendix B.

INPUT/OUTPUT STATEMENTS

Record Numbers

A record number may optionally follow the file designator as shown in READ statement forms 3, 5, 8, 10, 12, 14, 16, and 18. The record number must have a value from 1 to the number of records in the designated file, and the "=" separating the file designator and the record number may be replaced by an apostrophe (for EBCDIC-coded source input) or a quote mark (for BCL- or BCD-coded source input). The presence of a record number causes the file to be positioned at the specified record before the read is performed. Otherwise, the file position is unchanged prior to the read. After the READ statement is executed, the file is positioned at the record immediately after the record(s) read.

Use of Buffers

The intermediate storage between a program file and internal storage consists of the file buffers. The size of each file buffer is equal to the number of words in each file block. The size of a file block and the number of buffers associated with a file may be specified via the file declaration card as explained in appendix B.

When a READ statement is executed, the file buffers are checked to determine if any buffer contains the record to be read. If the record is found in a buffer, the data is accessed from the buffer and the file is not affected. If the record is not found in a buffer, then a read on the file block containing the record is initiated, placing the record in a buffer from which the data is then transferred to internal program storage.

Formatted READ Statement

A formatted READ statement is denoted by forms 1 through 10 and 19 through 22 of the READ statement. Such statements are always associated with a FORMAT statement, an array containing format specifications, a NAMELIST, or free-formatting, represented here by the item "f". When a FORMAT is associated with a READ statement, the data to be transferred is scanned and edited according to the designated format specifications. FORMAT statements and format specifications contained in arrays are discussed in chapter 13. Special input considerations are involved when a NAMELIST name is employed by a READ statement. (No I/O list may be used with such statements.) The NAMELIST concept is discussed in detail in chapter 14. A free-format item f allows the input record to contain the input values in free-format, separated by commas. Free-format is discussed later in this chapter. The following are examples of valid formatted READ statements:

```
READ (5,1) AB
READ (FILEN, 75) (A(I), I=3,10)
READ 12
READ (23, 12345,DATA=12400) 6,A,4,B,3,C
READ (23=12,AF)
READ 5,ARRAY
READ (5,ARRAY)
READ (5,B,RESULT=X,ERR=2)Y,Z(6),(A(J),J=1,6)
READ (I+2=IR-1,NMLST)
READ /,A,B
```

Unformatted READ Statement

An unformatted READ statement is denoted by forms 11 through 18, 23, and 24. No editing of transferred data is associated with these forms of the READ statement.

Execution of an unformatted READ statement causes the input of one logical record from the designated file. If an I/O list is specified as part of the statement, data will be transferred to the specified locations from the record. Transfer occurs as full 48-bit words, and the records accessed should have been generated by unformatted WRITE statements. If no I/O list is specified on an unformatted READ, one record will be skipped in the indicated file (i.e., the file position will be advanced one record).

The following is a sample of a program structure employing the unformatted READ statement.

```
      FILE  4=TEMP, UNIT=DISK
            .
            .
            .
            DO 1 I=1,10
            .
      1     WRITE(4)A,B,C
            .
            .
            REWIND 4
            DO 60 I=1,10
            READ(4)X,Y,Z
            .
            .
      60    CONTINUE
      99    STOP
            END
```

Internal Data Transfer

If an array name is specified instead of an expression designating a file number, the character string contained in that array is used as the "external string" for input editing. For example, the statements

```
      DIMENSION PACKED(5), UNPAKD(30)
      READ(PACKED,5)UNPAKD
    5 FORMAT (30C1)
```

take character data, stored six characters per word of the array PACKED, and place them one character per word in the right-hand position of the words of the array UNPAKD.

The following example shows how different kinds of data records might be mixed. If a card begins with the identification NAME, the data in columns 11-16 will be stored as characters in NAME. If the card begins with the identification NUMBER, the contents of positions 11-16 will be stored as the value of ID in integer form.

```
      REAL NAME, DATA(1)
      READ(5,5) TYPE,DATA
  5   FORMAT(A6, 4X, A6)
      IF(TYPE.EQ.'NAME')NAME=DATA(1)
      IF(TYPE.EQ.'NUMBER') READ(DATA,7)ID
  7   FORMAT(I6)
```

- **WRITE STATEMENT**

The executable WRITE statement allows data to be output to peripheral storage or an array from the internal storage locations indicated by the I/O list or format portion of the statement. The transferred data may be converted during the transfer process and positioned within records of a file as indicated by a specified FORMAT statement or format specification item, or written without editing if no format items are specified. Provision may also be made to handle errors incurred during the write.

The proper forms of the WRITE statement are:

1. WRITE (n,f)m 2. WRITE (n=r,f)m 3. WRITE (n,f,c)m 4. WRITE (n=r,f,c)m 5. WRITE (n,f) 6. WRITE (n=r,f) 7. WRITE (n,f,c) 8. WRITE (n=r,f,c) 9. WRITE (n)m 10. WRITE (n=r)m 11. WRITE (n,c)m 12. WRITE (n=r,c)m 13. WRITE (a,f)m 14. WRITE (a,f,c)m 15. WRITE (a,f) 16. WRITE (a,f,c) 17. WRITE (a)m 18. WRITE (a,c)m	where n and r are arithmetic expressions representing a file designator and record number, respectively; f is the label of a FORMAT statement, an array name denoting an array containing format specifications, a NAMELIST name (in forms 5-8), or a free-format designator (in forms 1-4 and 13-16); c is a result clause list; m is an I/O list; and a is an array identifier designating an internal data transfer. (See below.)

File Referenced

Execution of any WRITE statement causes data to be written from internal storage to one or more records of a program file or an array. The file referenced is indicated via the file designator, n. The default kinds of files referenced by various file designators are discussed in appendix B.

Record Numbers

A record number may optionally follow the file designator as shown in WRITE statement forms 2, 4, 6, 8, 10, and 12. The record number must have a value from 1 to the number of records in the designated file, and the "=" separating the file designator and the record number may be replaced by an apostrophe (for EBCDIC-coded source input) or a quote mark (for BCL- or BCD-coded source input).

The presence of a record number causes the file to be positioned at the specified record before the write is performed. Otherwise, the file position is unchanged prior to the write. After the WRITE statement is executed, the file is positioned at the record immediately following the record(s) written. The record pointed to is the default record to be accessed by the next input or output statement.

INPUT/OUTPUT STATEMENTS

Formatted WRITE Statement

A formatted WRITE statement is denoted by forms 1 through 8 and 13 through 16 of the WRITE statement. Such statements are always associated with a FORMAT statement, an array containing format specifications, a NAMELIST, or free-formatting, as represented here by the item "f". When a FORMAT is associated with a WRITE statement, the data to be transferred is converted and positioned within the record(s) according to the designated format specifications. FORMAT statements and format specifications contained in arrays are discussed in chapter 13. Special output considerations are involved when a NAMELIST name is employed by a WRITE statement as a format item. No I/O list may be used with such statements. The NAMELIST concept is discussed in detail in chapter 14. A free-format item causes values to be output in a free-field form, separated by commas or spaces. Free-format is discussed later in this chapter. The following are examples of valid formatted WRITE statements:

```
WRITE(6,/) A,B
WRITE(6,1) AB
WRITE (FILEN,75)(A(I),I=3,10)
WRITE(23+I,12345,ERR=12400) 6,A,4,B,3,C
WRITE(23=12,AF)
WRITE(7,ARRAY)
WRITE(6,B,RESULT=X,ERR=2)Y,Z(6),(A(J),J=1,6)
WRITE (I+2 = (IR*2)+1, NMLST )
```

Unformatted WRITE Statement

An unformatted WRITE statement is denoted by forms 9 through 12, 17, and 18. No editing of transferred data is associated with these forms of the WRITE statement.

Execution of an unformatted WRITE statement causes one logical record to be output to the indicated file. The mandatory I/O list denotes the sequence of values to be contained in the record. The contents of the indicated storage locations are placed unchanged in the generated record as full 48-bit words intended to be read by the unformatted READ statement. The I/O list may not be omitted on this type of WRITE statement.

The use of the unformatted WRITE statement is dependent upon the declared or default size of records within the affected file. When such a WRITE statement is executed which contains an I/O list calling for the transfer of more data words than may be contained on one logical record, then the program will be terminated, unless the statement contains a DATA action specifier, in which case transfer will occur to the statement indicated by the specifier. An unformatted WRITE statement may generate more than one physical record only if the file involved is a LINKWORD file. Such files may be

declared via a file declaration card bearing the LINKWORD file information clause (see appendix B). When an unformatted WRITE operation is performed on a LINKWORD file, a number of records are written on that file sufficient to contain the data words indicated by the I/O list. After this operation the file is positioned at the first record after the newly-generated record(s).

Internal Data Transfer

If an array name is specified instead of an expression designating a file number, the character string produced as the edited "output" (i.e., the "external string") is placed into the array. For example, the statements

```
      DIMENSION PACKED(5),UNPAKD(30)
      WRITE(PACKED,6) UNPAKD
    6 FORMAT(30C1)
```

take character data, stored one character per word in the right-hand portion of the words of the array UNPAKD, and store them six characters per word in the array PACKED.

Other examples of internal data transfer appear in the READ statement section of this chapter.

• PRINT STATEMENT

The executable PRINT statement allows data to be output to peripheral storage from the internal storage locations indicated by the I/O list or format portion of the statement. This statement is a variation of the formatted WRITE statement; no action clause list is allowed with this statement, and the file referenced is not explicitly specified.

The proper forms of the PRINT statement are:

1. PRINT f 2. PRINT f,m
where f is either the label of a FORMAT statement in this program unit, an array name denoting an array containing format specifications, a NAMELIST name (in form 1), or a free-format designation (in form 2), and m is an I/O list. (See preceding discussion.)

INPUT/OUTPUT STATEMENTS

Files Referenced

Execution of any PRINT statement causes data to be written from internal storage to one or more records of a program file. Every PRINT statement appearing within an executable program references the same file. If no file declaration card bears the identifier PRINTER (or PRINT) as the file designator and if no PRINTER compiler option card applies to the program during compilation, then the file referenced by the PRINT statement is the same file referenced by a READ, WRITE, or auxiliary I/O statement employing the file designator 6. The default kind of file associated with this designator is a line printer file; the internal name and default TITLE of the file is FILE6 (see appendix B).

The following is an example of a program compilation using the PRINT statement.

```
<I>COMPILE PROGRAM/ONE WITH FORTRAN; EBCDIC
 FILE  6=PRINTOUT/SINE
       PRINT 1,(SIN(I),I=1,10)
  1    FORMAT(F7.3)
       STOP
       END
<I>END
```

The program presented here generates output on a line printer file with the TITLE PRINTOUT/SINE.

If a file declaration card bearing the identifier PRINTER (or PRINT) as the file designator applies to the program during compilation, then the file referenced by the PRINT statement is not the file referenced by an I/O statement employing the file designator 6. The file accessed by the PRINT statement has the internal name PRINTER, and is a line printer file by default. The following is an example of a compilation deck employing a file declaration card which has this effect.

```
<I>COMPILE PROGRAM/TWO WITH FORTRAN
<I>FILE PRINTER(TITLE=ACTUAL/NAME); EBCDIC
 FILE  PRINTER=DUMMY/NAME,RECORD=6
       DO 2 COUNT=1,10
       A=SIN(COUNT)
       WRITE(6,4) A
  2    CALL PRINT(A)
  4    FORMAT(F7.3)
       STOP
       END
       SUBROUTINE PRINT(X)
       REAL F(3)/'('' VALUE = '',F6.3)'/
       PRINT F,X
       RETURN
       END
<I>END
```

This program generates two printer files: one with the internal name FILE6 and the default TITLE FILE6, and one with the internal name PRINTER and the TITLE ACTUAL/NAME. The PRINT statement references this latter program file.

Record Access

The position of the program file is unchanged prior to the execution of the PRINT statement. After such a statement is executed, the file is positioned at the record immediately after the record(s) written.

Data may be directed to specific records in a file through use of the BACKSPACE and REWIND auxiliary I/O statements prior to the appearance of the PRINT statement.

Format Considerations

A format item is associated with each PRINT statement, specifying the manner in which the transferred data is to be edited. This item may be the label of a FORMAT statement, the name of an array containing format specifications, a NAMELIST name, or a free-format designation. FORMAT statements and format specifications contained in arrays are discussed in chapter 13. When NAMELIST I/O is employed, the PRINT statement may contain no I/O list. The NAMELIST concept is discussed in detail in chapter 14. A free-format designation causes the indicated values to be printed in free-format, discussed later in this chapter.

• PUNCH STATEMENT

The executable PUNCH statement allows data to be output to peripheral storage from the internal storage locations indicated by the I/O list or format portion of the statement. This statement is a variation of the formatted WRITE statement; no action clause list is allowed with this statement, and the file referenced is not explicitly specified.

The proper forms of the PUNCH statement are:

1. PUNCH f 2. PUNCH f,m
where f is either the label of a FORMAT statement in this program unit, an array name denoting an array containing format specifications, a NAMELIST name (in form 1) or a free-format designation (in form 2), and m is an I/O list. (See preceding discussion.)

The operation of this statement is identical to the operation of the PRINT statement, except for the default file with which this statement is associated. If no file declaration card bearing the identifier PUNCH as the file designator applies to the program during compilation, then the file referenced by the PUNCH statement is the same file referenced by a READ, WRITE, or auxiliary I/O statement employing the file designator 7. The default kind of file associated with this designator is a card punch file; the internal name and default TITLE of the file is FILE7.

If a file declaration card bearing PUNCH as the file designator applies to the program during compilation, then the file referenced by the PUNCH statement is not the file referenced via the file designator 7. The file accessed by the PUNCH statement has the internal name PUNCH and is a card punch file by default.

The following are examples of valid PUNCH statements:

```
PUNCH   NMLST
PUNCH   1,(A(I),I=2,7)
PUNCH   ARRAYF, ARRAYX
PUNCH   12, I+2
PUNCH   /,G,R2
```

• BACKSPACE STATEMENT

The executable BACKSPACE statement is an auxiliary I/O statement which allows a program file to be repositioned before it is accessed by an input or output statement. When a BACKSPACE statement is executed, the indicated file is positioned one record before its present position, unless the file is presently positioned at its initial point, in which case no action is initiated.

The proper form of the BACKSPACE statement is:

BACKSPACE n
where n is an arithmetic expression, the truncated value of which is a file designator.

A possible use of the BACKSPACE statement is to allow the last record accessed by an input or output statement to be the next record accessed when such a statement is executed. The following sample program excerpt displays this use.

```
FILE  4 = SOURCE/FILE4, UNIT = DISK, RECORD = 4
      READ(4,1) A
      PRINT 2,A
      BACKSPACE 4
      READ(4,2) A
      PRINT 2,A
 1    FORMAT(A6)
 2    FORMAT(2X,A6)
      .
      .
      .
      STOP
      END
```

The initial statements of this program cause record 1 of the disk file SOURCE/FILE4 to be read into the variable A using two different formats and printed out each time. The BACKSPACE statement cancels the change of file position incurred by the first READ statement.

The BACKSPACE statement may not be used with the READER, PRINTER, or PUNCH file if such a file exists since this statement must employ a numeric file designator. (See the discussion of the READ statement and appendix B.) The BACKSPACE statement may not reference line printer, card punch, direct reader, or remote files.

- **CHANGE STATEMENT**

The executable CHANGE statement is an auxiliary I/O statement which may be used to modify the attributes of a file. The proper format of the CHANGE statement is

CHANGE (n,t)
where n is an arithmetic expression whose truncated value is used as a file designator and t is a list of file attribute assignment clauses (see below) separated by commas.

There must be at least one file attribute assignment clause. A file attribute assignment clause must be of the form fa = v, where fa is a valid file attribute identifier as listed in the Input/Output Subsystem Information Manual, form number 5000185, and v is an expression or mnemonic whose type and value are valid for that attribute identifier. Refer to the I/O Subsystem Manual for restrictions on when each attribute may be changed.

The CHANGE statement will cause the listed attributes to be set (assigned) to the corresponding values for the indicated file if none of the applicable restrictions have been violated.

Any pointer-valued attribute, such as TITLE, must be set using an array. The array must contain left-justified EBCDIC characters, 6 characters per word, and the string of characters must be terminated with a period. No compiler translation of character strings will be performed, even if compiler option BCL,

BCD, ASCII, or B5700 is set or the characters-per-word differs from 6 (via the CHARS compiler option). For example,

```
REAL A(2)/"MASTER/FILE."/
CHANGE (7,TITLE=A)
```

All logical-valued attributes must be explicitly set to a logical expression; for example,

```
CHANGE (7,OPTIONAL=.TRUE.)
```

• CLOSE STATEMENT

The executable CLOSE statement is an auxiliary I/O statement which allows a program file to be closed after the program has performed the necessary operations on it. The proper form of the CLOSE statement is:

1. CLOSE n 2. CLOSE (n) 3. CLOSE (n,DISP=DELETE) 4. CLOSE (n,DISP=KEEP)
where n is an arithmetic expression, the truncated value of which is a file designator.

The execution of a CLOSE statement causes the designated file to be returned to system control. The program no longer has access to the file.

If the file is a disk file generated by this program, the file will be closed and saved if the fourth form is used. (See the LOCK statement.) Use of the other forms will cause a new disk file to be immediately purged from the disk directory.

If the file is a card punch file, a card containing an endfile label is punched and the card punch device is released to the system.

If the file is a line printer file, the printer is skipped to channel 1, an ending label is printed, the printer is again skipped to channel 1, and the printer is released to the system.

If the file is a tape file and the third form is used, the file is closed and the tape is released to the system for other use. (See the PURGE statement.) The other forms of the CLOSE statement cause a tape mark (and an ending label for labeled tape) to be written after the last block on the tape (output tapes only). In any case, the tape is released to the system.

The first two forms of the CLOSE statement are equivalent. The third form is equivalent to the PURGE statement, and the last form is equivalent to the LOCK statement. DISP stands for DISPOSITION.

The following are examples of valid CLOSE statements:

```
CLOSE TAPENO
CLOSE 6
CLOSE (15, DISP=KEEP)
```

- **ENDFILE STATEMENT**

The executable ENDFILE statement is an auxiliary I/O statement which allows an endfile record to be written on an output file at the point at which the indicated file is positioned and the file to be closed. The proper form of the ENDFILE statement is:

ENDFILE n
where n is an arithmetic expression, the truncated value of which is a file designator.

The ENDFILE statement is intended primarily for use with tape files. When an ENDFILE statement follows an output statement on the file, an endfile record is written and the tape is positioned such that the next record written will follow the endfile record. When an ENDFILE statement follows a READ on the file, the tape is positioned at the start of the next file on the tape. When an ENDFILE statement follows a REWIND statement or another ENDFILE statement designating the same file, the ENDFILE statement is ignored.

When a program is employing multiple tape files on one physical reel (e.g., labeled files with TITLEs which consist of two identifiers separated by a slash, with the leftmost identifiers identical), then each file on this tape must have been closed via an ENDFILE statement before the next file may be written on the tape.

An ENDFILE statement executed on a tape file with a one-identifier TITLE or another kind of file causes the file to be closed and rewound. The file is not released to the system by this statement. An ENDFILE statement executed on a tape with a LABELTYPE of OMITTEDEOF will close the file and cause the tape to be positioned at the start of the next file on the tape.

The following is an excerpt of a sample main program employing the ENDFILE statement.

```
FILE  3=A60/A, UNIT=TAPE9, RECORD=20
FILE  4=A60/B, UNIT=TAPE9, RECORD=20
FILE  5=A60/C, UNIT=TAPE9, RECORD=30
      TAPENO=4
      FIND(TAPENO=1)
      PRINT 1
      WRITE(TAPENO,2,ERR=12) FCNA(TIME(1))
      .
      .
      .
      ENDFILE TAPENO
      TAPENO=TAPENO + 1
      WRITE(TAPENO,36,DATA=70) G, R1, R2
      .
      .
      .
      STOP
      END
```

- **FIND STATEMENT**

The executable FIND statement is an auxiliary I/O statement which provides a means of overlapping I/O time and processor time by positioning the designated file at the specified record and preloading that record into an available buffer in anticipation of a read or write on that record.

The proper form of the FIND statement is:

FIND(n = r)
where n and r are arithmetic expressions, the truncated values of which are a file designator and record number, respectively.

The record number must have a value from 1 to the number of records in the designated file, and the "=" separating the file designator and the record number may be replaced by an apostrophe (for EBCDIC-coded source input) or a quote mark (for BCL- or BCD-coded source input).

If, when a FIND statement is executed, the indicated record is not already in any of the program buffers, then if a free buffer is available, a read on the file block containing the appropriate record is initiated. If no free buffer is available, one is made available. The action initiated by the FIND statement may be carried out while the program carries out further computations prior to the input or output statement. This conserves execution time in many cases since the input or output statement must itself initiate this record access if it finds that no buffer contains the necessary record.

After this subsequent read or write action, the file is positioned by default at the record immediately following the record(s) read or written.

Error Handling

An error incurred during execution of a FIND statement will not cause the program to be terminated at that time. When a subsequent input or output action is initiated on the same file, the program will be terminated if the READ or WRITE statement does not contain a result clause list. If such a list is present, control will pass to an appropriate statement indicated by an action specifier in the list.

A FIND statement may follow another FIND statement without an intermediate READ. This would cause both records to be available in buffers if the file has at least two buffers.

The following sample program excerpt displays the use of the FIND statement.

```
FILE 4=A60/'FILE#2', UNIT=DISK, RECORD=10
     READ(4,20)A,B
     FIND(4=15)
     G=A*B**2
     READ(4,21,END=12,ERR=19) CNTRL
     .
     .
     STOP
     END
```

The initial statements of this program cause data to be read into the locations represented by the variables A and B from the first record of the disk file A60/FILE#2, the 15th record of this file to be placed into a buffer while the value of the variable G is calculated, and a value to be read from this record into the variable CNTRL. Control would immediately pass to the statement labeled 19 when this latter read is attempted if an error occurred during the execution of the FIND statement (such as an invalid record number).

• INQUIRE STATEMENT

The executable INQUIRE statement is an auxiliary I/O statement which may be used to determine the current status of file attribute information. The proper form of the INQUIRE statement is:

INQUIRE (n,t)
where n is an expression whose truncated value is used as a file designator and t is a list of file attribute clauses (see below) separated by commas.

There must be at least one file attribute clause, which must be of the form fa=v, where fa is a file attribute identifier as listed in the Input/Output

Subsystem Information Manual, form number 5000185, and v is a simple or subscripted variable whose type corresponds to that of the attribute.

Examples:

```
REAL KND,L(15),A(12)
LOGICAL OPN
INQUIRE (10,KIND=KND,OPEN=OPN,MAXRECSIZE=L(10),TITLE=A)
```

After execution of this INQUIRE statement, KND will contain the KIND of file 10 (e.g., 7 or VALUE(PRINTER), 1 or VALUE (DISK), etc.), OPN will be TRUE if file 10 is open or FALSE if file 10 is closed, L(10) will contain the maximum record size in characters or words (depending on the UNITS and EXTMODE attributes), and array A will contain the TITLE of the file in EBCDIC characters (six, 8-bit characters per word) terminated by a period.

All pointer valued attributes, such as TITLE, must be assigned to arrays into which the character strings will be transferred. The strings will be in 8-bit EBCDIC characters, regardless of the value of compiler options BCL, B5700, BCD, ASCII, and CHARS. The string is terminated by a period. Any array used in this manner must be long enough to contain the character string it is to receive. All character positions in the array not needed to store the attribute will be left unchanged.

For example:

```
FILE    70 (TITLE=MASTER/FYLE",KIND=DISK)
        REAL FNAME(3)/3*1/
        INQUIRE (70,TITLE/FNAME,KIND/KND)
```

After the execution of the INQUIRE statement, FNAME(1)="MASTER", FNAME(2) = "/FYLE.",FNAME(3)=1, and KND=VALUE (DISK), i.e., one.

• LOCK STATEMENT

The executable LOCK statement is an auxiliary I/O statement which allows a program file to be closed and saved after the program has performed the necessary operations on it. The proper form of the LOCK statement is:

LOCK n
where n is an arithmetic expression, the truncated value of which is a file designator.

The execution of a LOCK statement removes the designated file from the control of the program and saves it.

If the designated file is a tape file, the file is closed, the tape is rewound, and the system message "MT u RW/L" is displayed at the supervisory

console. (In this message, u is the unit number of the peripheral device.) The tape reel may then be removed.

If the designated file is a disk file, the file is removed from program control and is placed in the disk directory for the appropriate length of time as specified by its SAVEFACTOR attribute. A disk file generated by a FORTRAN program will not remain on disk unless that file is closed via a LOCK statement.

If the designated file is not a disk file, the unit is made inaccessible to the system until it is manually reset. This applies to devices which are accessed directly by the program but not to devices normally accessed via backup files (such as line printers and card readers).

For disk files, the LOCK statement is equivalent to CLOSE(n,DISP=KEEP).

The following are examples of valid LOCK statements:

```
LOCK TAPENO - 1
LOCK 22
LOCK I+J
```

- **OPEN STATEMENT**

The executable OPEN statement is an auxiliary I/O statement which causes a file to be explicitly opened. This action is not required. The file may not be already open when this statement is executed. Attributes may optionally be assigned to the file at the time the file is opened.

The proper forms of the OPEN statement are:

OPEN (n) OPEN (n,t)
where n is an arithmetic expression whose truncated integer value is used as a file designator and t is a list of file attributes assignment clauses (see below) separated by commas.

If the second form of the OPEN statement is used, t must be a list of file attribute clauses, separated by commas, of the form fa=e, where fa is a valid file attribute identifier as listed in the Input/Output Subsystem Information Manual, form number 5000185, and v is an expression or mnemonic whose type and value are valid for that attribute identifier. The use of these file attribute clauses will cause the listed attributes to be set (assigned) to the corresponding values for the indicated file as it is opened. Explicit use of the file attribute OPEN within the OPEN statement causes a warning diagnostic and the specified action is ignored.

Any pointer-valued attribute, such as TITLE, must be set using an array. The array must contain left-justified EBCDIC characters, 6 characters per word, and the string of characters must be terminated with a period. No compiler translation of character strings will be performed, even if compile option BCL, BCD, ASCII, or B5700 is set, or the characters-per-word differs from 6 (via the CHARS compiler option).

Example:

```
REAL A(2)/"MASTER/FILE."/
OPEN (7,TITLE=A)
```

All logical valued attributes must be explicitly set to a logical expression.

Example 1:

```
OPEN (7,OPTIONAL=.TRUE.)
```

Example 2:

```
REAL A(2)/"SOME/FILE."/
OPEN (30, KIND=DISK, TITLE=A,AREAS=2,MAXRECSIZE=10)
OPEN(9)
```

The OPEN statement may not be used with the default output printer file when using the batching facilities of the FORTRAN compiler.

• PURGE STATEMENT

The executable PURGE statement is an auxiliary I/O statement which allows a program file to be closed, purged from the system, and its space to be released for other use. The proper form of the PURGE statement is:

PURGE n
where n is an arithmetic expression, the truncated value of which is a file designator.

The following are examples of valid PURGE statements:

```
PURGE 10
PURGE FILENO
PURGE 9 +KFILE
```

A PURGEd tape file may be reused. PURGEing a card reader, card punch, line printer, or remote file simply closes the file.

• REWIND STATEMENT

The executable REWIND statement is an auxiliary I/O statement which allows a program file to be positioned at its initial point without releasing program control of the file. The proper form of the REWIND statement is:

REWIND n
where n is an arithmetic expression, the truncated value of which is a file designator.

The REWIND statement may be executed at any point within a program in order to position a program file at its initial point, causing the first record of the file to be the default record read or written when the next input or output statement is executed which references that file. The default record accessed by the first input or output statement in a program referencing a particular file is the first record of that file. Therefore, an initial REWIND statement is unnecessary.

If the file is positioned at its initial point when the REWIND statement is executed, no action is initiated.

When a REWIND statement is executed which references a file on a multi-file tape reel, the entire tape reel is rewound and positioned at beginning-of-tape.

The REWIND statement may not be used with line printer, card punch, card reader, or remote files.

The following is an excerpt of a sample program employing the REWIND statement.

```
      FILE 5=A60/D, UNIT=DISK,RECORD=10
           DO 1 I=1,100
           READ 5,A,B,C
           .
           .
           .
       1   CONTINUE
           FILENO=5
           .
           .
           .
           REWIND (FILENO)
           DO 601 COUNT=1,100
           .
           .
           .
      601  WRITE(5,2)X,Y,Z
      999  STOP
           END
```

This program reads the first 100 records of the disk file A60/D, performs data manipulation, rewinds the disk file, and rewrites the first 100 records of the file.

INPUT/OUTPUT STATEMENTS

FREE-FORMAT INPUT AND OUTPUT

The use of a free-format designator with the READ, WRITE, PRINT, and PUNCH statements allows I/O to be performed with editing, but without using a format statement. The appropriate format is selected automatically, but variations of the free-format designator give the user some control over the form of the output.

The general form for a free-format designator is:

ar/sw
where a is an optional asterisk (*), s is an optional second slash (/), and r and w are optional single precision arithmetic expressions enclosed in parentheses.

Input

On input, only the simplest form consisting of a single slash (/) should be used. It allows input from records consisting of data items separated by commas.

All blanks are ignored. Character strings must be enclosed by quote marks ("). Complex constants should be input as two single precision items.

Example:

If B is logical and C is complex and if the input record consists of:

1b2b3b,T,2,3

then the statement READ/,A,B,C assigns 123.0 to A, .TRUE. to B, and (2.0,3.0) to C.

Output

On output, each value is edited into an appropriate format. An edited item is never split across a record boundary. If the record is too short to hold any reasonable representation of the item, a string of pound signs (#) is output in place of the item.

Data items are normally separated by a comma and a space. If the optional second slash (/) is used, they are separated by two spaces. Note that output produced in this manner cannot be read by a free-format input statement.

If the optional asterisk is used, the name of the data item and an equal sign (=) are output prior to the value of the data item. If the data item is not a variable name, then <EXP> is output as the name of the data item.

If the optional r is present, the value of the expression in parentheses, truncated to an integer if necessary, specifies the maximum number of items that will be placed in one record.

If the option w is present, the value of the expression in parentheses, truncated to an integer if necessary, specifies the maximum of the field into which the data item is edited.

In the following examples, the output produced by each statement is shown after the statement.

```
PRINT/,123.456000,-.00123,1234,.TRUE.,'WOW'
123.456, -0.00123, 1234, T, WOW

PRINT (2)//,1234.567,-2.3,4,-1.23456E9,6D7
     1234.567            -2.3
           4      -1.23456E+9
       6.0D+7

WRITE(6,//(10))-1.234567E-14,5.4321D12345,.TRUE.
-1.2346-14  5.43+12345           T

PRINT(3)/(6),-12.34,.NOT..TRUE.,'TOO LONG','"WOW"',8.2D12345
     -12.34,       F, ######,
      "WOW", D12346,
PRINT */(10),X,SIN(X)
     X=0.7, <EXP>=0.64,
```

FILE HANDLING CAPABILITIES

A B 6700/B 7700 FORTRAN program may perform a large number of operations on program files via the available I/O statements. The forms of the possible types of program files are discussed in the Input/Output Subsystem manual (5000185). A brief description of the manner in which a FORTRAN program may manipulate these files is presented in this chapter.

Line Printer Files

Line printer files may be handled via the WRITE, PRINT, CLOSE, ENDFILE, LOCK, OPEN, CHANGE, INQUIRE, and PURGE statements. Spacing of file lines is controlled by the carriage control character contained in the leftmost character position of each output record. (See chapter 13.)

All printer files generated by a program are placed in backup disk files prior to their automatic printing and removal by the system if the run-time system option LPBDONLY is set. Line printer files may only be generated sequentially, i.e., no record number may be used in the WRITE statement.

INPUT/OUTPUT STATEMENTS

Card Reader Files

Card reader files may be handled via the READ statement and all auxiliary I/O statements except the ENDFILE statement. Card files are read by the card reader device and placed in pseudo card disk files by the system. These pseudo card files are the files which are accessed by a program, thus allowing the files to be positioned similar to disk files.

Card Punch Files

Card punch files may be handled via the WRITE, PUNCH, CLOSE, ENDFILE, LOCK, CHANGE, OPEN, INQUIRE, and PURGE statements. The TITLE of each card punch file is punched on the initial label card of each file.

Card punch files may only be generated sequentially, i.e., no record number may be used in the WRITE statement.

Disk Files

Disk files may be handled via any I/O statement. Such files, once generated, may be used both for input and output and may be handled serially or randomly, depending on the absence or presence, respectively, of record numbers in the I/O statements.

More than one disk file may be open at one time.

A disk file generated by a program must be closed via the LOCK statement before the file will be retained in the disk directory.

Tape Files

Tape files may be handled via any I/O statement. Such files, once generated, may be used both for input and output (depending on the presence of a write ring), and may be handled serially or randomly, depending on the absence or presence, respectively, of record numbers in the I/O statements.

Tape program files may be classified according to whether the file is the only program file accessed from the reel on which it resides or is one of several program files accessed from the same tape reel.

Single-File Tape Reels

The TITLE of a program file which is the only program file accessed from that tape may be any valid TITLE. A SAVEFACTOR should be assigned to the file in order to prevent premature purging of the tape by the system.

When an ENDFILE or REWIND statement is executed on such a tape file, the file is closed and rewound, but its space will not be allocated to another program file requiring a tape.

The tape file will be identified in system messages as "MT m", where m is the unit number of the tape device on which the reel is mounted.

Multi-File Tape Reels

One tape reel may contain more than one tape file used in the same program. Such files must have TITLEs consisting of two suitable identifiers separated by a slash, with one multifile ID (i.e., identical leftmost identifiers).

Thus, the following could be TITLEs of program files contained on one tape reel:

```
A60/ONE
A60/DATA
A60/CODE
```

As files are being generated on one tape reel, each file written must be closed via an ENDFILE statement before the next file to be generated may be written. The last file generated should be closed by a CLOSE or REWIND statement. When an ENDFILE statement is encountered as a tape file is being written, an end-file record is written to terminate the file, and the tape is positioned at the next available space in anticipation of the generation of another tape file.

CHAPTER 13

FORMAT SPECIFICATIONS AND STATEMENTS

This chapter discusses the format specifications provided to indicate data conversion during input and output and the method of associating these specifications with input and output statements via the FORMAT statement and program arrays containing such specifications.

FORMAT SPECIFICATIONS

The following format specifications are provided to allow conversion of data between the representation of such data as characters in peripheral storage and the configurations of this data in internal storage:

```
 Aw
 Cw
 Dw.d
 Ew.d
 Fw.d
 Gw.d
wHs or 's' or "s"
 Iw
 Jw
 Lw
 Ow
fP
 Tt
 U or Uw or Uw.d
 Vw or Vw.d
wX or X
 Zw
```

In addition, there are two format specification modifiers:

```
K
$
```

In the list of specifications and in the following paragraphs, w and t represent unsigned non-zero literals denoting the width of the field in the external character string and a character position relative to the beginning of the external record, respectively. Item d represents an unsigned integer literal denoting the number of digits in the fractional part of the external string. Item f represents an integer literal scale factor. Item s represents a string of 264 or fewer characters.

The internal storage types referred to in this chapter are discussed in chapter 11.

FORMAT SPECIFICATIONS AND STATEMENTS

• A Format Specification

The alphanumeric format specification Aw causes data to be transferred to or from internal storage as EBCDIC characters.

Input Using Aw

On input, the Aw specification causes the character string of width w in the external field to be assigned to the corresponding simple variable or array element in the I/O list.

Characters are stored internally in 8-bit EBCDIC form, six characters per data word. The Aw format specification allows characters to be placed left-justified into a data word with trailing blanks. If the field width, w, exceeds 6 (12 for double precision variables), the rightmost six (12) characters in the input field are stored into the corresponding variable in the I/O list and the remaining characters are ignored. The indicated variable may be of type REAL, INTEGER, or DOUBLE PRECISION.

Examples:

		INTERNAL VALUE	
EXTERNAL STRING	SPECIFICATION	Real or Integer	Double Precision
ABCD	A4	ABCDbb	ABCDbbbbbbbb
bABCDb	A6	bABCDb	bABCDbbbbbb
ABCDEFGHI	A9	DEFGHI	ABCDEFGHIbbb
bABCDEFGHIJKb	A13	GHIJKb	ABCDEFGHIJKb

Output Using Aw

On output, the Aw specification causes the characters contained in the appropriate variable in the I/O list to be placed into an external string of length w. If w is greater than 6 (12 for double precision variables), the 6 (12) characters contained in the data word will be placed right-justified in the field, preceded by blanks. If w is less than or equal to 6 (12 for double precision variables), the left-most w characters in the data word will be placed in the field. If one of the character fields in the word contains a bit pattern which does not correspond to an EBCDIC graphic, then a question mark will be placed in that position.

Examples:

INTERNAL VALUE	SPECIFICATION	EXTERNAL STRING
ABCbbX	A3	ABC
ABCbbX	A6	ABCbbX
ABCbbX	A9	bbbABCbbX
ABCDEFGHIJKb	A9	ABCDEFGHI
ABCDEFGHIJKb	A13	bABCDEFGHIJKb

- **C Format Specification**

The Cw format specification has the same effect as the Aw format specification except for the following:

a. Characters are placed into and taken from the right-hand portion of a word (rather than the left-hand portion).

b. On input the unused portion of the word is filled with zeros (rather than blanks). Since Z00 does not represent an EBCDIC character on output, a question mark will be used.

Examples (Z00 is represented by a ?):

		INTERNAL VALUE	
EXTERNAL STRING	SPECIFICATION	Real or Integer	Double Precision
ABCD	C4	??ABCD	????????ABCD
bABCDb	C6	bABCDb	??????bABCDb
ABCDEFGHI	C9	DEFGHI	???ABCDEFGHI
bABCDEFGHIJKb	C13	GHIJKb	ABCDEFGHIJKb

INTERNAL VALUE	SPECIFICATION	EXTERNAL STRING
??ABCD	C4	ABCD
bABCDb	C4	BCDb
??ABCD	C9	bbb??ABCD
??bbbABCDEFG	C4	DEFG
??bbbABCDEFG	C11	?bbbABCDEFG
??bbbABCDEFG	C13	b??bbbABCDEFG

• D Format Specification

The double precision format specification Dw.d causes data appearing in an external character string as a double precision constant to be associated with an internal storage location for purposes of input or output.

Input Using Dw.d

On input, the Dw.d specification causes the value of the double precision constant written with or without exponential notation in a string of w input characters to be assigned to the corresponding I/O list element.

The Dw.d specification allows the input constant to contain as many decimal places as desired by use of the decimal place count, d. If no decimal point appears in the input string, a decimal point is implied as specified by d. Thus, the input string 100D0 when read using the specification D5.2 would be interpreted as the double precision constant 1.D+0 with two implied decimal places in the input string. A decimal point is assumed d places from either the right edge of the input field or from the E denoting the exponent, if there is one.

The field width, w, must be greater than or equal to the specified number of decimal places, d. A blank is interpreted as a zero.

The Dw.d specification functions similar to the Ew.d specification with the extended ability to accept double precision constants written in exponential notation rather than real constants written in exponential notation. Real constants not employing this notation may be input via this format specification.

Output Using Dw.d

On output, the Dw.d specification causes the value of the corresponding item in the I/O list to be written as an output character string of length w, representing a double precision constant expressed in exponential notation. The exponent is adjusted so that the decimal point is positioned as specified by the decimal place count, d.

The specified width of the output field, w, must be greater than or equal to the number of specified decimal places, d, plus 10. This provides for a 7-character exponent part, a decimal point, a zero preceding the decimal point, and a sign. If this rule is violated, the field will be filled with asterisks.

The Dw.d specification is essentially equivalent to the Ew.d specification except for the presence of a D rather than an E in the exponent part of the output string and a five-digit rather than a two-digit exponent.

- **E Format Specification**

The real format specification Ew.d causes data appearing in an external character string as a real constant to be associated with an internal storage location for purposes of input or output.

Input Using Ew.d

On input, the Ew.d specification causes the value of the real constant written with or without exponential notation in a string of w input characters to be assigned to the corresponding I/O list element.

The Ew.d specification allows the input constant to contain as many decimal places as desired by use of the decimal place count, d. If no decimal point appears in the input string, a decimal point is implied as specified by d. Thus, the input string 100E0 when read using the specification E5.2 would be interpreted as the real constant 1.E+0 with two implied decimal places in the input string. A decimal point is assumed d places from either the right edge of the input field or from the E denoting the exponent, if there is one.

The field width, w, must be greater than or equal to the specified number of decimal places, d. A blank is interpreted as a zero.

Examples:

EXTERNAL STRING	SPECIFICATION	INTERNAL VALUE
bbbbbb25046	E11.4	+2.5046
bbbbb25.046	E11.4	+25.046
-bb25046E-3	E11.4	-0.0025406
bb250.46E-3	E11.4	+0.25046
b-b25.04678	E11.4	-25.04678

Output Using Ew.d

On output, the Ew.d specification causes the value of the corresponding item in the I/O list to be written as an output character string of length w, representing a real constant expressed in exponential notation. The exponent is adjusted so that the decimal point is positioned as specified by the decimal place count, d.

The specified width of the output field, w, must be greater than or equal to the number of specified decimal places, d, plus 7. This provides for a 4-character exponent part, a decimal point, a zero preceding the decimal point, and a sign. If this rule is violated, the field will be filled with asterisks.

Examples:

INTERNAL VALUE	SPECIFICATION	EXTERNAL STRING
+36.7929	E13.5	bb0.36793Eb02
-36.7929	E12.5	-0.36793Eb02
-36.7929	E11.5	**********

- **F Format Specification**

The real format specification Fw.d causes data appearing in an external character string as a real constant to be associated with an internal storage location for purposes of input or output.

Input Using Fw.d

On input, the Fw.d specification causes the value of the real constant written with or without exponential notation in a string of w input characters to be assigned to the corresponding I/O list element.

The decimal point may be positioned as indicated in the input string or located as desired via the decimal place count, d. If no decimal point appears in the input string, a decimal point is implied as specified by d. A decimal point is assumed d places from the right edge of the input field. Thus, the input string 1234 when read using the specification F4.2 would be interpreted as the real constant 12.34 with two implied decimal places in the input string.

The field width, w, must be greater than or equal to the specified number of decimal places, d, and must include the decimal point and exponent field when either or both are present.

Examples:

EXTERNAL STRING	SPECIFICATION	INTERNAL VALUE
36725931	F8.4	+3672.5931
3.672593	F8.4	3.672593
-367259.	F8.4	-367259
-3672.E2	F8.4	-367200
367259E2	F8.4	+3672.59
3.672E-1	F8.4	+.3672
367259	F6.6	+0.367259
b-b3456	F7.2	-34.56

Output Using Fw.d

On output, the Fw.d specification causes the value of the corresponding item in the I/O list to be written as an output character string of length w, representing a real constant expressed without using exponential notation. The decimal point is adjusted such that d digits follow the decimal point.

The constant is right-justified over blanks within the field, and the specified width of the output field, w, must be greater than or equal to the number of specified decimal places, d, plus 1. The possible presence of a minus sign for a negative datum must be taken into consideration when specifying the field width.

The internal value is rounded to satisfy the decimal point specification, and the field will contain asterisks if the value to be output has an integer part too large for the allotted field.

Examples:

INTERNAL VALUE	SPECIFICATION	EXTERNAL STRING
+36.7929	F7.3	b36.793
+36.7934	F9.3	bbb36.793
-0.0316	F6.3	-0.032
0.0	F6.4	0.0000
0.0	F6.2	bb0.00
+579.645	F6.2	579.65
+579.645	F4.2	****
-579.645	F6.2	******

- **G Format Specification**

The general format specification Gw.d is a generalized floating-point conversion which may be used with I/O list elements of type REAL, INTEGER, or DOUBLE PRECISION. This specification is interpreted as F, E, or D, depending upon the type and the magnitude of the value of the list element. The output string will have a length of w characters, regardless of the value being read or written.

On input, the Gw.d specification accepts real or double precision constants.

On output, the Gw.d specification produces an F, E, or D format representation of the value of the list element according to the following criteria. If v is the absolute value of the list item, then for k=0,1,2,..., d, if $10^{d-k-1} \leq V < 10^{d-k}$, then format F(W-4).(d-k),4X is used. For V<.1 or $V \geq 10^{d}$, then EW.d is used. A scale factor affects the format only if Ew.d is used.

Example: The statements

```
    DO5K = -1,B
    V = .5*10**(2-K)
  5 WRITE(6,6)V
  6 FORMAT(2P,X,G10.2)
```

produce the output

```
    50.00E+01
      50.
      5.0
     0.50
    50.00E-03
```

- **H Format Specification (Strings)**

The string (or Hollerith) format specification wHs or 's' (for EBCDIC-coded source input or "s" for BCL- or BCD-coded source input) allows character strings to be input or output without employing program storage locations. The item s in this specification is a string of one to 264 characters. To represent an apostrophe within a proper string when using EBCDIC-coded source input, two immediately adjacent apostrophes must be used. Similarly, a quote mark in a proper string must be represented by two immediately adjacent quote marks when BCL- or BCD-coded source input is employed.

The field width, w, must reflect the number of characters in the string.

Input Using wHs or 's'

When employed as a format specification connected with a READ statement, the wHs or 's' specification causes the character string s to be replaced by the character string of the same length as that string appearing at the location on the input record referenced by this specification. The format specification is thus modified at the time of the program execution; use of this specification for a subsequent output action will result in the new string being output.

Example:

The following program excerpt demonstrates the use of the string format specification to accept two characters strings from the first record of a disk file and subsequently print them.

```
FILE  5=OXL/'367', UNIT=DISK, RECORD=10, BLOCKING=3
 1    FORMAT('INPUT STRING ONE  ',A6,'INPUT STRING TWO  ')
      READ 1,A
      IF(A.EQ.'ORDER1') A='MASTER'
      PRINT 1,A
      .
      .
      .
```

The execution of the READ statement will cause the strings 'INPUT STRING ONE ' and 'INPUT STRING TWO ' to be replaced by the first through eighteenth and the 25th through 42nd characters, respectively, in the first record of the disk OXL/367. These strings are written via the PRINT statement, with the first character in the first character string functioning as a carriage control character in the usual manner.

Output Using wHs or 's'

When the string format specification is used for output, the character string, s, will be written in the indicated position of the output record.

Example:

The execution of this program excerpt causes the string 'START LIST' to be printed.

```
REAL A(3)/'(X,10HSTART LIST)'/
WRITE(6,A)
```

- **I Format Specification**

The integer format specification Iw causes the external character string of width w to be associated with the corresponding I/O list element for purposes of data transfer.

Input Using Iw

On input, the Iw specification causes the value of the integer constant right-justified in the input field to be assigned to the corresponding I/O list variable. Blank characters are interpreted as zeros.

Examples:

EXTERNAL STRING	SPECIFICATION	INTERNAL VALUE
567	I3	+567
bb-329	I6	-329
-bbbb27	I7	-27
27bbb	I5	+27000
b-bb234	I7	-234

Output Using Iw

On output, the Iw specification causes the value of the corresponding I/O list element to be printed as an integer constant in a field of width w. The constant is right-justified over a field of blanks, and the plus sign is not printed for non-negative quantities.

If the value of the I/O list item requires a field larger than w, then w asterisks will be printed.

Floating-point values are rounded to an integer value before printing.

Examples:

INTERNAL VALUE	SPECIFICATION	EXTERNAL STRING
+23	I4	bb23
-79	I4	b-79
+67486	I5	67486
-67486	I5	*****
+978	I1	*
0	I3	bb0
+3.6	I2	b4

- **J Format Specification**

The integer format specification Jw causes the external character string of at most w characters to be associated with the corresponding I/O list element for purposes of data transfer.

Input Using Jw

On input, the Jw specification functions identically to the Iw specification.

Output Using Jw

On output, the Jw specification causes the value of the corresponding I/O list element to be printed as an integer constant in the minimum field necessary to contain the value without exceeding w. The plus sign is not printed for non-negative quantities.

If the value to be printed requires more than w characters, w asterisks will be printed.

Floating-point values are rounded to an integer value before printing.

Examples:

INTERNAL VALUE	SPECIFICATION	EXTERNAL STRING
+23	J5	23
-23	J5	-23
+233	J3	233
-233	J3	***
0	J3	0

- **L Format Specification**

The logical format specification Lw causes the logical value indicated by the contents of a character string of width w to be associated with the corresponding I/O list for purposes of data transfer.

Input Using Lw

On input, the Lw specification causes the corresponding LOGICAL element of the I/O list to be assigned the value .TRUE. or .FALSE., depending upon the contents of the field of width w. If the left-most non-blank character is a T, the variable will be assigned the value .TRUE.; otherwise, the value .FALSE. will be assigned. An all-blank field will yield the value .FALSE..

Example:

EXTERNAL STRING	SPECIFICATION	INTERNAL VALUE
T	L1	.TRUE.
bbF	L3	.FALSE.
bbbTRU	L6	.TRUE.
b	L1	.FALSE.

Output Using Lw

On output, the Lw specification causes the output string of width w to contain an indication of the logical value of the corresponding list element. If the logical value of this element is .TRUE. or .FALSE., a T or F, respectively, will be placed right-justified in this string. The remaining characters of the string will be blanks.

The I/O list element need not be a LOGICAL item. If bit 0 of the corresponding storage word is on or off, the logical value of the item is .TRUE. or .FALSE., respectively. (See the discussion of logical variables in chapter 11.)

Examples:

INTERNAL VALUE	SPECIFICATION	EXTERNAL STRING
.FALSE.	L1	F
.FALSE.	L3	bbF
.TRUE.	L2	bT
2	L1	F
3	L3	bbT

- **O Format Specification**

The octal format specification Ow causes the string of octal digits in the w-character field to be interpreted as an octal value and associated with the corresponding I/O list element for purposes of data transfer.

Input Using Ow

On input, the value represented by the octal digits in the input field is assigned to the corresponding list variable. Leading, trailing, and embedded blanks are interpreted as zeros. A minus sign (-) causes bit 46 of the storage word allocated to the variable to be complemented. (See chapter 11.) If the list element is single precision and the input datum is less than 16 octal digits long, it is stored right-justified in the storage word. If the list element is double precision, then the input datum is stored right-justified in the first word and zeros are stored in the second word if the input datum is 16 or fewer digits; if it is longer than 16 digits, the first 16 digits are stored in the first word and the remaining digits are stored right-justified in the second word. The maximum octal value which can be read is 7777777777777777 (77777777777777777777777777777777 for double precision variables).

Examples:

EXTERNAL STRING	SPECIFICATION	INTERNAL VALUE
16	O2	O0000000000000016
1777777777777777	O16	O1777777777777777
-16	O3	O2000000000000016
1234b56	O7	O0000000001234056
77b	O3	O0000000000000770

Output Using Ow

On output, the Ow specification causes the value of the specified data item to be printed as a string of octal digits, right-justified over blanks in a field of w characters. If w does not exceed 16 (32 for double precision variables), the contents of the right-most w*3 bits of the storage word are printed as a string of w octal digits. Such output never contains a printed sign.

Examples:

INTERNAL VALUE	SPECIFICATION	EXTERNAL STRING
00005677701234444	05	34444
00005677701234444	016	0005677701234444
00005677701234444	018	bb0005677701234444
9	02	11

- **P Format Specification**

The P specification may be used to modify the scaling of numbers input and output using the D, E, F, and G format specifications. A scale factor designator has the form

fP
where f is an integer constant which is the scale factor.

The scale factor affects all subsequent D, E, F, and G formats in a FORMAT statement until a new scale factor is encountered. A specification list may be considered to begin with a scale factor of zero.

A scale factor fP has the following effect with F format:

(external quantity) = (internal value) *10**f

With E and D format on input, a scale factor has no effect if the external field contains the character E or D, and has the same effect as with the F format if the external field does not contain an E or D.

With Ew.d and Dw.d format on output, the internal and external values are equal, but a scale factor affects the positioning of the decimal point in the field of significant digits and the value of the exponent is adjusted appropriately as follows:

a. If $f \leq 0$, then $|f|$ zeros are placed to the right of the decimal point followed by $d-|f|$ significant digits.
b. If $f \geq 0$, then f significant digits are placed to the left of the decimal point and d significant digits will be placed to the right of the decimal point.

A scale factor affects a G format specification only if the value is written using an E format (see the G format specification).

Examples:

EXTERNAL STRING	SPECIFICATION	INTERNAL VALUE
12345	2P, F5.1	12.345
123.4	2P, F5.1	1.234
12345	2P, E5.1	12.345
123.4	2P, E5.1	1.234
1.2E4	2P, E5.1	1.2E4
123D4	2P, D5.1	1.23D5

INTERNAL VALUE	SPECIFICATION	EXTERNAL STRING
1.234	2P, F10.2	bbbb123.40
1.234	-2P, F10.2	bbbbbb0.01
1.234	0P, E10.2	bbb.12E+01
1.234	2P, E10.2	b12.34E-01
1.234	-1P, D10.2	bbb.01D+02

- **T Format Specification**

The tab specification Tt allows a tab to the position in the record to be affected by the next format specification. The item t is the number of the character (from the left edge of the record) which is to be the start of the next external input or output string.

Example:

```
      X1 = 1.7
      X2 = 2.3
      WRITE(6,6) X1, SIN(X1), X2, SIN(X2)
    6 FORMAT(T20, 'X', T35 'SIN(X)'/(2F20.2))
      END
```

This program produces the following output:

```
                   X                                SIN(X)
                1.70                                  0.99
                2.30                                  0.75
```

The T specification may be used to reread a portion of an input record or write over a portion of a record before output as illustrated by the following example:

```
      READ(5,5) N1,N2
    5 FORMAT (T6, I3, T7, I2)
      WRITE(6,6) N1, N2
    6 FORMAT(T5,1HN/('b**********',T5,J3))
      END
```

If a data card contains 123456789 in columns 1-9, then this program produces the following output:

```
      N
***678****
***78*****
```

- **U Format Specification**

The U editing specification causes the data item to be output in a form best suited for the item. Real, integer, and double precision items are output in a format that combines readability with maximum numerical significance. Logical items are output at T or F and occupy one character position in the record. Character strings are treated as real. If the number of characters required to edit the item is greater than the number left in the current record, the record is output and the item placed in the next record, with a blank inserted in position 1 of printer and remote files.

The form Uw is similar to U, with the added restriction that the edited item may not exceed w characters. If the data item cannot be edited into a field of w characters, a field of w asterisks is output.

The form Uw.d is similar to Uw, with the added restriction that the total field width occupied by the edited item may not be less than d characters. In this case, the number of non-blank characters (those representing the data item itself) may not exceed w characters. Thus, if d>w, d-w leading blanks will be inserted.

Example: The statements

```
     X=12.3456
     WRITE(6,6).FALSE., 'XXXXXX',X,X,X,X
  6  FORMAT(X,U,X,U,X,U,X,U3,X,U5.3,X,U3.5)
```

produce the output

Fb-0.0126838235294b12.3456b12.b12.35bbb12.

- **V Format Specification**

Refer to the section on run-time format editing, discussed later in this chapter.

- **X Format Specification**

The editing specification wX will cause w characters to be skipped on input and will cause a field of w blank characters to be inserted into the output record. The specification X has the same effect as 1X.

- **Z Format Specification**

The hexadecimal format specification Zw causes the string of hexadecimal digits in the w-character field to be interpreted as a hexadecimal value and associated with the corresponding I/O list element for purposes of data transfer.

FORMAT SPECIFICATIONS AND STATEMENTS

Input Using Zw

On input, the value represented by the hexadecimal digits in the input field is assigned to the corresponding list variable. Leading, trailing, and embedded blanks are interpreted as zeros. A minus sign (-) causes bit 46 of the storage word allocated to the variable to be complemented. (See chapter 11.)

If the list element is single precision and the input datum is less than 12 hexadecimal digits long, it is stored right-justified in the storage word. If the list element is double precision, then the input datum is stored right-justified in the first word and zeros are stored in the second word if it is 12 or fewer digits; if it is more than 12 digits, the first 12 digits are stored into the first word and the remaining digits are stored right-justified in the second word. The maximum hexadecimal value which can be read is FFFFFFFFFFFF (FFFFFFFFFFFFFFFFFFFFFFFF for double precision variables).

Examples:

EXTERNAL STRING	SPECIFICATION	INTERNAL VALUE
6F	Z2	Z00000000006F
1FFFFFFFFFFF	Z12	Z1FFFFFFFFFF
-16	Z3	Z400000000016
1234b568	Z8	Z000012340568
FFCb	Z4	Z00000000FFC0
C1C2C3C4C5C6	Z12	ZC1C2C3C4C5C6 or 'ABCDEF'

Output Using Zw

On output, the Zw specification causes the value of the specified data item to be printed as a string of hexadecimal digits, right-adjusted over blanks in a field of w characters. If w does not exceed 12 (24 for double precision variables), the contents of the right-most w*4 bits of the storage word are printed as a string of w hexadecimal digits. Such output never contains a printed sign.

Examples:

INTERNAL VALUE	SPECIFICATION	EXTERNAL STRING
Z0000E5551010	Z5	51010
Z0000E5551010	Z12	0000E5551010
Z0000E5551010	Z16	bbbb0000E5551010
32	Z2	20
'123456'	Z12	F1F2F3F4F5F6

FORMAT SPECIFICATION MODIFIERS

Format specification modifiers may be placed immediately to the left of a format specification used to edit a data item for output. If a repeat count is used, it should be to the left of any modifiers used. More than one modifier may be used with a format specification. A modifier may not be used on input.

For example, 2KF10.3 and 8K$G20.6 are valid, but $2J5 is not.

• K Format Specification Modifier

The K modifier causes commas to be inserted between digit triples to the left of the decimal point.

• $ Format Specification Modifier

The $ modifier causes the character $ to be inserted immediately to the left of an edited item.

Examples:

INTERNAL VALUE	SPECIFICATION	EXTERNAL STRING
17.347	$F10.2	bbbb$17.35
-1234567	KI10	b1,234,567
-1234567	K$G15.2	bbbb$-1,234,567
1234567.11111	KF15.5	1,234,567.11111

THE FORMAT STATEMENT

The nonexecutable FORMAT statement specifies what type of data conversion and editing is to be performed during input or output between the characters in the records in program files and the data words in internal storage.

The proper form of the FORMAT statement is:

n FORMAT(s)
where n is a statement label and s is a specification list which consists of appropriate format specifications modified as desired by parentheses, repeat counts, and specification modifiers, separated by commas and/or one or more slashes.

The FORMAT statement is associated with a formatted READ, a formatted WRITE, a PRINT, or a PUNCH statement via the statement label, n. Such statements may also specify data conversion and editing to be performed via a NAMELIST name (as described in chapter 14), via a free-format item (producing or accepting free-format values), or via an array containing a parenthesized format list. The latter format item is discussed later in this chapter.

FORMAT SPECIFICATIONS AND STATEMENTS

Correspondence is established between I/O list elements and the elements of the format item list in the normal FORTRAN fashion. Repetition of a portion of the format items is accomplished by enclosing this format portion in parentheses and optionally preceding the left parenthesis by a repeat count indicating the number of times the format portion is to be traversed before using the succeeding item.

If there is an I/O list on the I/O statement referencing the FORMAT statement, the specification list must contain at least one specification besides wHs, 's', Tt, wX, X, P, slash, or parentheses.

Record Fields

A field in a formatted record contains an EBCDIC character string representation of the value of a single I/O list element. The field width of a format specification specifies the number of characters in the field. A field begins with the first character position of a record for the first I/O list element, or the first character after the previous field. The Tt specification may be used to change the character pointer for subsequent editing.

In input numeric fields, all blank characters are interpreted as zeros. Plus signs are optional on input and may be omitted. When input data is entered under a real format specification (i.e., D, E, F, or G), a decimal point appearing in the input field overrides the decimal place count, d.

Complex I/O List Elements

A complex I/O list element is considered, for purposes of I/O conversion, as two real list elements. Thus, there must be two format items (or a repeated specification) for every complex I/O list element.

Format Field Separators

Two fields in a format item list are separated by a comma, a slash, or a series of slashes. A slash is used to indicate the end of a record. On input, any remaining characters in the current record are ignored when a slash is encountered in the specification list. On output, the construction of the current record is terminated and any subsequent output is placed in the next output record(s). Multiple slashes may be used to skip several records of input or generate several blank records on output. The final right parenthesis of a format also acts to indicate the end of the current record.

Repeat Counts

Format specifications and format list portions enclosed in parentheses may optionally be immediately preceded by an unsigned nonzero integer constant. This constant indicates the number of times that portion of the specification list is to be interpreted. If no such repeat count is indicated, a repeat count of 1 is assumed.

If the outer right parenthesis of the format specification list is encountered before the I/O list is exhausted, control reverts to the repeat count (if present) of the repeat specification group terminated by the last preceding right parenthesis. If no other right parenthesis exists in the specification list, then control reverts to the first left parenthesis of the specification list.

The following are proper examples of the use of repeat counts. In each case, the repeat count is 3.

```
3F10.4
3(A6/)
3(3A6,3(/I12)/)
```

Run-Time Format Editing

If an asterisk (*) appears in a format specification list in place of a field width (w), a decimal place count (d), scale factor (s), or a tab position (t), then the I/O list will be accessed once and the value of the I/O list element obtained will be used to replace the *. A new I/O list element is required each time an * is encountered in the specification list.

For example, this program:

```
REAL A(2)/'(I3,2X,I*)'/
PRINT A,(I,I+1,10**I,I=0,7)
STOP
END
```

produces this output:

```
0  1
1  10
2  100
3  1000
4  10000
5  100000
6  1000000
7  10000000
```

In this program, the value of I+1 in the I/O list is used to replace the asterisk in the specification list. (See later paragraphs in this chapter for a discussion of the use of arrays to contain format specification lists.)

If a V appears in a format specification list in place of the letter, A, C, D, E, F, G, I, J, L, O, or Z identifying a format specification, then the I/O list will be accessed and the contents of the right-most eight bits of the data word referenced will be treated as an EBCDIC character to be used in the actual format specification. If the right-most eight bits represent the blank character (Z40), then the character represented by the left-most eight bits will be used. This character may be only one of the letters just listed; V is not acceptable. A new I/O list element is required each time the V is encountered in the specification list.

The proper character may be placed into the I/O list element by being read using the Cw format specification or other means.

For example, since the right-most eight bits of the integer constant 193 represent the EBCDIC letter A, this program:

```
      DATA B/'TEXT'/
      PRINT 1,6,'A      ',B,193,B
1     FORMAT(I2,X,V4,/X,V4)
      STOP
      END
```

will produce this output:

```
 6 TEXT
TEXT
```

Carriage Control

Any record which is intended to be output upon a line printer must contain a prescribed carriage control character in its first character position. This character will not be printed but is interpreted by the printing routines as follows:

CHARACTER	EFFECT
blank	Single space
0	Double space
1	Page eject
+	No line feed (overprint)
n (digit 2 thru 9)	Advance to the channel n

Any other character in the first character position will cause the output line to be moved right one character; the first character will be printed. In this case, single spacing will be assumed.

All carriage spacing indicated by the carriage control character is performed prior to printing the line.

Format Specifications in Arrays

A formatted READ, WRITE, PRINT, or PUNCH statement may employ an array name as a format item. This array must contain an EBCDIC string of characters representing a parenthesized format specification list. This specification list is defined identically as the specification list portion of a FORMAT statement.

The specification list may be stored in the array via an initial value list, a READ statement, or assignment statements. Employing a READ statement allows the format specification list to be unspecified until execution time.

Example:

The following program illustrates the use of format specifications contained in arrays.

```
<I>COMPILE "FORMAT ARRAY" WITH FORTRAN; EBCDIC
      DIMENSION FORM(2), DATA(6)
      DATA FORM/'(X,F6.3)'/, DATA/1.2,3.4,5.6,7.8,9.2,4.5/
      DO 1 I=1,3
   1  WRITE(6,FORM) DATA(I)
      READ(5,2) FORM
   2  FORMAT(2A6)
      DO 3 I=4,6
   3  WRITE(6,FORM) DATA(I)
      STOP
      END
<I>EBCDIC FILE5
 (X,F6.1)
<I>END
```

This program produces the following output:

```
1.200
3.400
5.600
  7.8
  9.2
  4.5
```

A second example appears in chapter 12, under the heading INPUT/OUTPUT STATEMENTS.

CHAPTER 14

NAMELIST STATEMENTS AND NAMELIST I/O

The NAMELIST statement is a nonexecutable statement which causes a list of program variables and arrays to be associated with a unique symbolic name called a NAMELIST name. This NAMELIST name may only be employed in place of the format specifier in a READ, WRITE, PRINT, or PUNCH statement to produce the input or output in free-field format of selected data items from among the items in the associated NAMELIST list.

The proper form of the NAMELIST statement is:

NAMELIST/n/p ...
where n is a NAMELIST name which is constructed in the same manner as a variable name, p is a NAMELIST list which is a list of variables and array names separated by commas, and the ellipsis (...) denotes as many repetitions of the clause "/n/p" as desired with different NAMELIST names and lists.

A NAMELIST name may only be used in place of a format specifier and must be declared as such by a NAMELIST statement appearing in the program unit in which it is used. This I/O statement may not contain an I/O list. The NAMELIST name may not be a dummy argument nor may the elements of a NAMELIST list be dummy arguments if the NAMELIST appears in a subprogram. This name may be used for no other purpose.

The NAMELIST statement must follow any declarations and type statements referencing the elements of the NAMELIST list(s).

One NAMELIST statement may be used to associate several NAMELIST names with NAMELIST lists. For example, if A, B, and C are array names, the NAMELIST statement:

```
NAMELIST/NMLST1/A,G/NMLST2/A,B,A2/LIST/C,XR,S,T
```

associates the array A and the variable G with the NAMELIST name NMLST1, associates the arrays A and B and the variable A2 with the NAMELIST name

NMLST2, and associates the array C and the variables XR, S, and T with the NAMELIST name LIST. The values of A and G could subsequently be input by a statement of the type:

READ NMLST1 or READ(5,NMLST1)

INPUT USING NAMELIST

Values may be input to internal storage locations from peripheral storage via formatted READ statements employing NAMELIST names in place of format specifiers. No I/O lists may appear on such statements. (See chapter 12.) When such a READ statement is executed, the referenced file is accessed repeatedly, beginning at its present position, until an appropriate NAMELIST input record (as described below) is found or an end-of-file condition occurs. NAMELIST input records must have the following format:

a. A NAMELIST record may consist of one or more physical records.

b. The physical records of the NAMELIST record are free format except for the first two characters in the first physical record. The first character of this record is ignored, while the second character must be an ampersand (&). The appropriate NAMELIST name must be the first item following the &. This name is followed by at least one blank, which is followed by the list of value assignments.

c. Each NAMELIST contains a list of value assignments, separated by commas. A value assignment is either a variable followed by an equal sign (=) and a literal value (in that order) or an array name or array element followed by an equal sign which is followed by one or more literal values separated by commas, with or without repeat specifications. Each variable and array referenced in this list must have been associated with the NAMELIST name in a NAMELIST statement.

d. The literal values contained in the value assignments are assigned to the indicated variables and array elements when a READ statement is executed referencing the appropriate NAMELIST name. These literal values may be numeric literals or Hollerith or proper string literals containing one to six characters. (See chapter 4.) If assignment is being made to an array name or array element, the literal(s) may be immediately preceded by a repeat specification of the form: n*, where n is an unsigned nonzero integer constant.

e. If assignment is being made to an array name, only one literal value or as many values as there are elements in the array may be indicated by the value list. If assignment is being made to an array element, only one literal value or as many values as there are succeeding elements in the array may be indicated by the value list. Values are assigned to the elements of the array in the order in which they are stored internally, beginning with the indicated element (the first element if only the array name is used). (See chapter 11.)

f. The NAMELIST input must be terminated by the appearance of "&END". The remaining contents of the physical record on which this item appears will be ignored.

After the physical records bearing a NAMELIST record are read, the file is positioned at the next record in the file in anticipation of the next I/O statement referencing that file.

Data items associated with the NAMELIST name which do not have values assigned to them in the NAMELIST record will retain their present values.

Example

The following compile-and-go deck illustrates the use of the NAMELIST concept for data input. (<I> denotes an invalid character in card column 1.)

```
<I>COMPILE NAMELIST/TEST WITH FORTRAN;EBCDIC
 C ** INPUT USING NAMELIST **
       DIMENSION A(2,2),B(3)
       NAMELIST /NTEST/A,B,C,D
       DATA B(2)/6/
       READ 1,C
  1    FORMAT(I1)
       FIND(5'C)
       READ 2,C
  2    FORMAT(F3.1)
       READ NTEST
       READ 1,B(3)
       PRINT 3,A,B,4,C,D
  3    FORMAT(I2/2(A6/),F4.1/3(I2/),F*.1/X,A3)
       STOP
       END
```

```
<I>EBCDIC FILE5
 3
 3.2
 6.4
 9.6
 2.0
 &NTEST A=1,2*'ABCDEF',1,B=7,
 D=3HIJK&END   (END OF NTEST RECORD)
 5
<I>END
```

When this program is executed with the indicated input data deck, the first record of FILE5 is accessed and the value 3 is input to the variable C. The file is then positioned at the record indicated by the value of C (record 3), and the value contained on this record (6.4) is input to C. When the READ NTEST statement is executed, FILE5 is accessed repeatedly until the NTEST NAMELIST record is located, at which time the value assignments indicated in this record are made. Since the names A,B, and D had been associated with the NAMELIST name NTEST by a preceding NAMELIST statement, the value assignments indicated are valid. The values specified are assigned to the elements of the array A as if these assignment statements had been executed in this order:

```
A(1,1)=1
A(2,1)='ABCDEF'
A(1,2)='ABCDEF'
A(2,2)=1
```

The value 7 is assigned to array element B(1), and the value 'IJK ' is assigned to the variable D. The note "(END OF NTEST RECORD)" is ignored. Another read is performed on this file and the value 5 is read from the record immediately following the NAMELIST record into the array element B(3).

Therefore, the printed output generated by the PRINT statement is:

```
1
ABCDEF
ABCDEF
1.0
7
6
5
6.4
IJK
```

OUTPUT USING NAMELIST

Values may be output to peripheral storage from internal storage locations via output (i.e., formatted WRITE, PRINT, and PUNCH) statements employing NAMELIST names in place of format specifiers. No I/O list may appear on such a statement. (See chapter 12.) When such an output statement is executed, the value of every variable and array element associated with the NAMELIST name is placed in a generated NAMELIST record in the referenced file; the record is suitable for input by a READ statement employing that NAMELIST name and is constructed of one or more physical records.

The generated NAMELIST record is of the form discussed in the preceding section. No repeat specifications will appear in the output, and the values of array elements will be output for every element of the array in the order in which the elements are stored internally. (See chapter 11.)

Values are output according to the declared or default type of each data item. No facilities are available for the writing of strings using NAMELIST output.

The following program excerpt employs NAMELIST output.

```
DIMENSION A(4,4),M(10),N(20)
NAMELIST /NAMEA/A,D,K,M,N,X /NAMEB/M,N,X
.
.
.
WRITE(6,NAMEA)
.
.
.
PUNCH NAMEB
.
.
.
```

USING NAMELIST WITH SEPARATE COMPILATIONS

A syntax error is given if a global variable (i.e., an initialized variable, a variable in COMMON storage, or a variable affected by the OWN compiler option) appears as an element of a NAMELIST list in a subprogram if the SEPARATE option is set for the compilation.

CHAPTER 15

SUBPROGRAMS AND INTRINSICS

Subprograms are program units which may be invoked in the main program, another subprogram, or the same subprogram as a separate executable procedure, or which may be used for data initialization.

SUBROUTINE SUBPROGRAMS

A subroutine is a sequence of statements initiated by a SUBROUTINE statement and terminated by an END statement.

SUBROUTINE Statement

The nonexecutable SUBROUTINE statement indicates the beginning of a subroutine subprogram and may specify the dummy arguments employed in that subprogram.

The proper forms of the SUBROUTINE statement are:

SUBROUTINE s or SUBROUTINE s (d)
where s is a SUBROUTINE name, and d is a list of dummy arguments, separated by single commas.

SUBROUTINE Names

A SUBROUTINE name is constructed in the same manner as a variable name. No type is associated with this name, and once a symbolic name is used as a SUBROUTINE name it may be used for no other purpose in that program.

Dummy Argument Lists

Each element of a dummy argument list may be a simple variable name, an array name, a FUNCTION or SUBROUTINE name, or an asterisk (*). A variable in such a list may optionally be enclosed in slashes (/).

A dummy argument may be a simple variable name, in which case the actual argument in the subroutine call must be a variable name, an array element, or an expression. The actual argument must be of the same type as the dummy argument if the dummy argument is neither REAL nor INTEGER type.

If an array appears as a dummy argument, the corresponding actual argument may be an array or array element. In the latter case, the actual argument may be thought of as an array whose first element is that array element. Array handling between actual and dummy argument lists is discussed in the section on the CALL statement in chapter 8. When a dummy argument is an array, that array may have adjustable dimensions in the declarator in the subprogram, but such dimensions may only be variables appearing in the dummy argument list or an element of a COMMON block which appears in that subprogram. Thus, the following two statements reflect proper usage of these constructs:

```
SUBROUTINE SUBA(A,N)
REAL A(N,6)
```

Care should be taken that the dummy array is not dimensioned larger than the corresponding actual array which may result in an invalid index condition when the dummy array is referenced.

An asterisk in a dummy argument list reserves that position in the list for a statement label preceded by an ampersand (&) in the actual argument list; this argument is used in connection with nonstandard RETURN statements, described in chapter 8.

The following are valid examples of SUBROUTINE statements:

```
SUBROUTINE SUBA(/X/)
SUBROUTINE MULTMT(ARRAY1,ARRAY2,*)
SUBROUTINE MAX(A,B,C,/D/,E,*,F)
```

Use of Subroutines

The executable statements in a subroutine body are executed when the subroutine is invoked by a CALL statement. The subroutine may cause program termination by executing a STOP or CALL EXIT statement or may cause control to subsequently return to the calling program unit via the RETURN statement. There is always an implied RETURN statement preceding the END statement, so that if the execution sequence attempts to go beyond the last executable statement of a subroutine, the effect is the same as executing a RETURN statement.

Arrays passed as arguments to a subroutine are always called by name. The subroutine is thus passed the starting address of the array and thereby accesses the actual array. Any array elements which have their values changed in the subroutine will retain these new values when control is returned to the calling program unit. Thus, storage is not allocated to the array in the subroutine but at the point in the program when the array was originally declared.

Simple variables and array elements are also called by name in that any change in the final value of the corresponding dummy argument will be assigned to the actual argument when execution of the subroutine is completed. All other arguments are called by value in that only the value of the argument is passed to the subroutine. A simple variable or array element will be passed by value if the argument is preceded by a unary +.

If a dummy argument is enclosed in slashes, any change in the value of the dummy argument in the subroutine will be recorded simultaneously as the value of the actual argument.

Consider the following program:

```
COMMON X
CALL  S(X)
PRINT/,X
END
SUBROUTINE S(/A/)
COMMON X
A=2; X=3
RETURN
END
```

The value printed for X is 3.0. However, if the dummy argument A were not enclosed in slashes, the value printed for X would be 2.0, since the value of the dummy argument A is 2.0 upon completion of the subroutine. At that point, it is assigned as the value of the actual argument X, changing its value from 3.0 to 2.0.

FUNCTION SUBPROGRAMS AND STATEMENT FUNCTIONS

FORTRAN functions are procedures which return a single value to a calling program unit at the point at which they are referenced. A function may consist of a program unit similar to a SUBROUTINE subprogram, called a FUNCTION subprogram or may be declared as a single expression via a statement function declaration appearing in the program unit in which it is referenced.

Function Subprogram

A function subprogram is a sequence of statements initiated by a FUNCTION statement and terminated by an END statement.

FUNCTION Statement

The nonexecutable FUNCTION statement indicates the beginning of a function subprogram and specifies the dummy arguments employed in that subprogram to obtain the function value.

The proper forms of the FUNCTION statement are:

FUNCTION f(d) or t FUNCTION f(d)
where f is a FUNCTION name, d is a list of dummy arguments separated by single commas, and t is a type as defined in chapter 7.

FUNCTION Names

A FUNCTION name is constructed in the same manner as a variable name. A type is associated with this symbolic name which indicates the type of the value returned when the function is referenced. Default types are assigned to FUNCTION names in the same manner as variable and array names. FUNCTION names beginning with the letters A through H and O through Z and $ are assigned the default type of REAL, and FUNCTION names beginning with the letters I through N are assigned the default type INTEGER.

The default type associated with a FUNCTION name may be altered via the item t in the FUNCTION statement. For example, the function L$OR will be of type LOGICAL if it is initiated by a FUNCTION statement of the form:

```
LOGICAL FUNCTION L$OR (A)
```

When a function is of a non-default type, then the FUNCTION name must appear in an appropriate type statement or be affected by an appropriate IMPLICIT statement in every program unit in which that function is referenced. For example, a program unit referencing the preceding function must contain a statement of the sort:

```
LOGICAL L$OR
```

Once a symbolic name has been used as a FUNCTION name, it may be used for no other purpose in that program. Such a name may appear on the left side of an equal sign in an assignment statement only in the body of the function subprogram. The value to be returned must be assigned to the FUNCTION name in this manner before a RETURN is executed.

Dummy Argument Lists

The list of dummy arguments in a FUNCTION statement is defined in exactly the same manner as the same portion of a SUBROUTINE statement. Every FUNCTION statement must contain at least one dummy argument.

Example of Function Subprogram

The following sample function subprogram, RSHIFT, returns a word identical to the word passed to the dummy argument A with each bit in the word shifted over N bit positions. If N is greater than 0, a right circular shift is performed. If N is less than 0, a left circular shift is performed.

```
      FUNCTION RSHIFT(A,N)
C     CIRCULAR RIGHT SHIFT N BITS FOR N>0
C     CIRCULAR LEFT SHIFT -N BITS FOR N<0
      M = MOD(N,48)
      RSHIFT = A
      IF(M.EQ.0) RETURN
      IF(M.LT.0) M = M+48
C     MOVE RIGHT-MOST M BITS OF A INTO RSHIFT
      RSHIFT = CONCAT(RSHIFT,A,47,M-1,M)
C     MOVE REMAINING 48-M BITS OF A INTO RSHIFT
      RSHIFT = CONCAT(RSHIFT,A,47-M,47,48-M)
      RETURN
      END
```

The CONCAT intrinsic is described later in this chapter.

Statement Functions

A statement function is a function which may be expressed as one statement. It has the same general form as an assignment statement except that the FUNCTION name and dummy argument list appear to the left of the replacement operator. This statement is called a statement function declaration.

Statement Function Declaration

A statement function declaration is a nonexecutable statement of the form:

f(d)=e
where f is a FUNCTION name, d is a list of dummy arguments separated by single commas, and e is an expression.

A statement function is local to the program unit in which the statement function declaration appears. It is referenced in that program unit exactly like any other function and may not be referenced by any other program unit.

The name of a statement function may not appear in an EXTERNAL statement. A warning message is issued by the compiler if a statement function declaration follows the first executable statement in the program unit.

Function Type

The type of the value returned when a statement function is referenced depends upon the type associated with the FUNCTION name. A default type is associated with this name in the same manner as a variable name. This default type may be changed by the appearance of the FUNCTION name in a type statement or the implicit typing of the name in an IMPLICIT statement in the same program unit.

For example, the following is a valid combination of statements:

```
IMPLICIT LOGICAL (L)
LFCN(A,B)=A.IS.B
```

When this statement function is referenced, the 48 bits in words A and B will be compared, and if their states are identical, the function will return the value .TRUE.. The expression in a statement function may be a logical expression only if the FUNCTION name is of LOGICAL type.

Dummy Argument Lists

The list of dummy arguments in a statement function may contain only simple variable names (without slashes).

Use of Functions

A function is executed when a function reference is encountered in an expression. The value obtained from the function execution is returned to the point of the function reference.

A function reference has the form: f(a), where f is the FUNCTION name (or an ENTRY name), and a is a list of actual arguments.

Execution of a function subprogram begins with the first executable statement following the FUNCTION statement.

When a FUNCTION subprogram is referenced, the actual argument list may contain the same items as an actual argument list of a CALL statement. When a statement function is referenced, the actual argument list may contain only expressions.

If a function reference occurs in an I/O list, the execution of that function may not cause any I/O statement to be executed which references the same file as the I/O statement bearing the I/O list.

A nonstandard RETURN executed within a function subprogram will result in no value being returned from the function reference. Execution will resume with the statement bearing the indicated label. (See the discussion of the RETURN statement in chapter 8.)

Array handling in function subprograms is performed in the same manner as for subroutines; and a variable name enclosed in slashes will be called by name in the same manner as for subroutines.

RECURSION

A subroutine or function subprogram may be defined "recursively" by containing a reference to itself or to another subprogram that may directly or indirectly reference it.

Example (Towers of Hanoi): N disks, all of different sizes, are placed on a pole (call it pole 1) in order of their size, with the largest at the bottom. The disks are to be moved to pole 3, using an additional pole 2, subject to the following rules:

a. Only one disk may be moved at a time.
b. No disk may ever be placed on a pole above a smaller disk.

The following program prints the moves to be made for N=4 disks.

```
      DATA N/4/
      CALL HANOI(N,1,2,3)
      END
C
      SUBROUTINE HANOI(N,N1,N2,N3)
      IF(N.GT.1) CALL HANOI(N-1,N1,N3,N2)
      PRINT//,"MOVE DISK",N,"FROM",N1,"TO",N3
      IF(N.GT.1) CALL HANOI(N-1,N2,N1,N3)
      RETURN
      END
```

ENTRY STATEMENT

A subroutine or function subprogram may optionally contain one or more non-executable ENTRY statements which allows execution to begin at a point other than the first executable statement in the subprogram when the subprogram is invoked.

The proper forms of the ENTRY statement are:

ENTRY p or ENTRY p(d)
where p is an ENTRY name and d is a list of dummy arguments separated by single commas.

SUBPROGRAMS AND INTRINSICS

ENTRY Names

An ENTRY name is constructed in the same manner as a variable name. A type is associated with such a name only if the entry appears in a function subprogram. Once a symbolic name has been used as an ENTRY name it may be used for no other purpose in that program.

Dummy Argument Lists

The dummy argument list of an ENTRY statement is constructed identically to the same portion of a SUBROUTINE statement.

Entry in Subroutine Subprograms

An ENTRY statement appearing in the body of a subroutine subprogram provides an alternate point at which to begin execution when the subroutine is called. Execution begins with the first executable statement after the ENTRY statement. The associated ENTRY name is considered to be a SUBROUTINE name.

The dummy argument list in an ENTRY statement is independent of any other dummy argument lists in the program unit. If the same names appear in more than one dummy parameter list, they need not occupy the same positions in those lists. An ENTRY statement may contain more dummy arguments than the SUBROUTINE statement. Entry into a subroutine at any point results in initialization of only those dummy arguments included in the dummy argument list of the statement at which entry occurs. Of the remaining arguments, those which are simple variables are reset to zero, and those which are arrays are nonexistent for this call. Local variables are reset to zero or retain the values assigned them by initial value lists in the normal manner. See compiler options OWN and OWNARRAYS, chapter 19.

The following is an example of a subroutine containing an ENTRY statement:

```
SUBROUTINE SUBA(/A/,B)
ENTRY XSUBA(/B/,A)
A=A**B
B=A
RETURN
END
```

This subroutine may be called by statements of both these forms:

```
CALL SUBA(G,H)
CALL XSUBA(G,H)
```

Entry in Function Subprograms

An ENTRY statement appearing in the body of a function subprogram provides an alternate point at which to begin execution when the function is referenced.

Execution begins with the first executable statement after the ENTRY statement. The associated ENTRY name is considered to be a FUNCTION name.

The dummy argument list is defined as for an ENTRY statement in a subroutine subprogram.

The ENTRY name must be referenced in a type or IMPLICIT statement both in the calling program unit and in the subprogram body if the name is to have a non-default type.

All ENTRY names in a function subprogram are equivalenced to the FUNCTION name; thus, the value to be returned from the function reference may be assigned to an ENTRY name or a FUNCTION name before a RETURN statement is executed.

The following is an example of a function subprogram containing an ENTRY statement:

```
FUNCTION QSIN(X)
A=2.3
ENTRY RSIN(X)
QSIN=1.1 + A + SIN(X)*2
RETURN
END
```

This function may be referenced, for example, in either of these ways:

```
IF(QSIN(B).LT.1.5) GO TO 7
IF(RSIN(B).LT.1.5) GO TO 7
```

BLOCK DATA SUBPROGRAMS

A BLOCK DATA subprogram is a sequence of statements initiated by a BLOCK DATA statement and terminated by an END statement. The purpose of this subprogram is to allow elements of COMMON storage to be assigned initial values.

BLOCK DATA Statement

The nonexecutable BLOCK DATA statement indicates the beginning of a BLOCK DATA subprogram. The proper form of this statement is:

```
BLOCK DATA
```

Use of BLOCK DATA

The BLOCK DATA subprogram may contain no executable statements. Its sole purpose is to specify initial values for one or more blocks of COMMON storage by means of DATA statements or initial value lists on type statements.

There may be as many BLOCK DATA subprograms as desired in an executable program. None may be used with separate compilations.

SUBPROGRAMS AND INTRINSICS

A BLOCK DATA subprogram may contain the following statements in the order indicated.

- BLOCK DATA statement.
- IMPLICIT statement.
- Type statements.
- COMMON statements.
- DATA statements.
- END statement.

The following is a valid example of a BLOCK DATA subprogram:

```
BLOCK DATA
IMPLICIT LOGICAL (L)
COMMON L1,LCNTRL/BLOCKA/A(3)
DATA L1,LCNTRL,A/2*.TRUE.,3*1.2/
END
```

INTRINSICS

Intrinsics may be considered to be function subprograms which are known to the compiler and need not be supplied in the program. An intrinsic may, however, be redefined by providing a function subprogram with the same name as the intrinsic.

Table 15-1 lists the intrinsic names recognized by the B 6700/B 7700 FORTRAN compiler. The type of the value returned by the intrinsic reference is indicated as are the number and types of actual arguments passed to the intrinsic. ("DOUBLE" indicates DOUBLE PRECISION type.) A brief description of the function performed by each intrinsic is also given. In such descriptions, A1 and A2 represent the first and second arguments passed to the intrinsic, respectively. Multiple arguments must always be separated by single commas; such arguments must agree in number and type with the specification, with real and integer types being interchangeable.

The following mnemonics are used to denote the kind of each intrinsic:

IL - indicates that the intrinsic is an in-line function (i.e., code is placed in the code file at the point of the intrinsic reference to perform the indicated function) -- when the name of such an intrinsic appears in an EXTERNAL statement, the intrinsic will not be expanded in-line;

(n) - where n is an integer, indicates the number by which the intrinsic is identified;

MCP - indicates that the value returned is supplied by the Master Control Program;

CT - indicates that the intrinsic is a special compile-time procedure which produces a value during compilation.

Table 15-1. B 6700/B 7700 Intrinsics

INTRINSIC NAME	INTRINSIC TYPE	ARGUMENT TYPE(S)	NO. OF ARGUMENTS	FUNCTION PERFORMED	INTRINSIC KIND
ABS	REAL	REAL	1	Absolute value	IL
AIMAG	REAL	COMPLEX	1	Obtain imaginary part	IL
AINT	REAL	REAL	1	Largest integer ≤ |A1|	IL
ALGAMA	REAL	REAL	1	Log gamma function	(36)
ALOG	REAL	REAL	1	Natural logarithm	(18)
ALOG10	REAL	REAL	1	Common logarithm	(22)
AMAX0	REAL	INTEGER	≥2	Choose maximum value	IL
AMAX1	REAL	REAL	≥2	Choose maximum value	IL
AMIN0	REAL	INTEGER	≥2	Choose minimum value	IL
AMIN1	REAL	REAL	≥2	Choose minimum value	IL
AMOD	REAL	REAL	2	Remaindering: A1 (mod A2)	IL
AND	REAL	REAL	2	48-bit logical product	IL
ARCOS	REAL	REAL	1	Arccosine	(2)
ARSIN	REAL	REAL	1	Arcsine	(9)
ATAN	REAL	REAL	1	Arctangent	(3)
ATANH	REAL	REAL	1	Hyperbolic arctangent	(83)
ATAN2	REAL	REAL	2	Arctangent of A1/A2	(35)
CABS	REAL	COMPLEX	1	Absolute value	(31)
CCOS	COMPLEX	COMPLEX	1	Trigonometric cosine	(12)
CDABS	COMPLEX	REAL	1	Absolute value	(31)
CDCOS	COMPLEX	COMPLEX	1	Trigonometric cosine	(12)
CDEXP	COMPLEX	COMPLEX	1	Exponential: e**A1	(24)
CDLOG	COMPLEX	COMPLEX	1	Natural logarithm	(26)
CDSIN	COMPLEX	COMPLEX	1	Trigonometric sine	(21)
CDSQRT	COMPLEX	COMPLEX	1	Square root	(33)
CEXP	COMPLEX	COMPLEX	1	Exponential: e**A1	(24)
CLOG	COMPLEX	COMPLEX	1	Natural logarithm	(26)
CMPLX	COMPLEX	REAL	2	Create complex = (A1,A2)	IL
COMPL	REAL	REAL	1	48-bit logical complement	IL
CONCAT	REAL	(See text)	5	Partial word manipulation	IL
CONJG	COMPLEX	COMPLEX	1	Complex conjugate	IL
COS	REAL	REAL	1	Trigonometric cosine	(20)
COSH	REAL	REAL	1	Hyperbolic cosine	(5)
COTAN	REAL	REAL	1	Trigonometric cotangent	(39)
CSIN	COMPLEX	COMPLEX	1	Trigonometric sine	(21)
CSQRT	COMPLEX	COMPLEX	1	Square root	(33)
CTIME	(See text)	REAL Constant	1	Special compile-time TIME function	CT
DABS	DOUBLE	DOUBLE	1	Absolute value	IL
DACOS	DOUBLE	DOUBLE	1	Arccosine	(84)
DASIN	DOUBLE	DOUBLE	1	Arcsine	(85)
DATAN	DOUBLE	DOUBLE	1	Arctangent	(27)
DATAN2	DOUBLE	DOUBLE	2	Arctangent of A1/A2	(28)
DBLE	DOUBLE	REAL	1	Type conversion	IL
DCMPLX	COMPLEX	REAL	2	Create complex = (A1,A2)	IL
DCONJG	COMPLEX	COMPLEX	1	Complex conjugate	IL
DCOS	DOUBLE	DOUBLE	1	Trigonometric cosine	(41)
DCOSH	DOUBLE	DOUBLE	1	Hyperbolic cosine	(86)
DERF	DOUBLE	DOUBLE	1	Error function	(87)
DERFC	DOUBLE	DOUBLE	1	1/DEFF	(88)

Table 15-1. B 6700/B 7700 Intrinsics (Cont)

INTRINSIC NAME	INTRINSIC TYPE	ARGUMENT TYPE(S)	NO. OF ARGUMENTS	FUNCTION PERFORMED	INTRINSIC KIND
DEXP	DOUBLE	DOUBLE	1	Exponential: e**A1	(13)
DFLOAT	DOUBLE	INTEGER	1	Type conversion	IL
DGAMMA	DOUBLE	DOUBLE	1	Gamma function	(89)
DIM	REAL	REAL	2	A1 - min(A1,A2)	IL
DLGAMA	DOUBLE	DOUBLE	1	Log gamma function	(90)
DLOG	DOUBLE	DOUBLE	1	Natural logarithm	(15)
DLOG10	DOUBLE	DOUBLE	1	Common logarithm	(19)
DMAX1	DOUBLE	DOUBLE	≥2	Choose maximum value	IL
DMIN1	DOUBLE	DOUBLE	≥2	Choose minimum value	IL
DMOD	DOUBLE	DOUBLE	2	Remaindering: A1 (mod A2)	IL
DSIGN	DOUBLE	DOUBLE	2	(sign of A2)*|A1|	IL
DSIN	DOUBLE	DOUBLE	1	Trigonometric sine	(8)
DSINH	DOUBLE	DOUBLE	1	Hyperbolic sine	(91)
DSQRT	DOUBLE	DOUBLE	1	Square root	(34)
DTAN	DOUBLE	DOUBLE	1	Trigonometric tangent	(92)
DTANH	DOUBLE	DOUBLE	1	Hyperbolic tangent	(93)
EQUIV	REAL	REAL	2	48-bit logical equivalence	IL
ERF	REAL	REAL	1	Error function	(37)
ERFC	REAL	REAL	1	1/ERF	(94)
EXP	REAL	REAL	1	Exponential: e**A1	(32)
FLOAT	REAL	INTEGER	1	Type conversion	IL
GAMMA	REAL	REAL	1	Gamma function	(6)
HFIX	INTEGER	REAL	1	Type conversion	IL
IABS	INTEGER	INTEGER	1	Absolute value	IL
IDIM	INTEGER	INTEGER	2	A1 - min(A1,A2)	IL
IDINT	INTEGER	DOUBLE	1	Type conversion	IL
IFIX	INTEGER	REAL	1	Type conversion	IL
INT	INTEGER	INTEGER	1	Largest integer ≤ |A1|	IL
ISIGN	INTEGER	INTEGER	2	(sign of A2)*|A1|	IL
MAX0	INTEGER	INTEGER	≥2	Choose maximum value	IL
MAX1	INTEGER	REAL	≥2	Choose maximum value	IL
MIN0	INTEGER	INTEGER	≥2	Choose minimum value	IL
MIN1	INTEGER	REAL	≥2	Choose minimum value	IL
MOD	INTEGER	INTEGER	2	Remainder: A1 (mod A2)	IL
OR	REAL	REAL	2	48-bit logical sum	IL
RANDOM	REAL	REAL	1	Pseudo-random number generator	(7)
REAL	REAL	COMPLEX	1	Obtain real part	IL
SIGN	REAL	REAL	2	(sign of A2)*|A1|	IL
SIN	REAL	REAL	1	Trigonometric sine	(25)
SINH	REAL	REAL	1	Hyperbolic sine	(14)
SNGL	REAL	DOUBLE	1	Type conversion	IL
SQRT	REAL	REAL	1	Square root	(1)
TAN	REAL	REAL	1	Trigonometric tangent	(23)
TANH	REAL	REAL	1	Hyperbolic tangent	(30)
TIME	INTEGER	(See text)	1	Time function	MCP

The FORTRAN compiler will recognize other intrinsic names besides those just listed if the compiler option INSTALLATION is set for the compilation, the intrinsics file presently in use by the system contains the referenced intrinsic, and the intrinsic may be used by FORTRAN. Non-in-line and non-MCP intrinsics correspond to code in the intrinsics file, and it is this code which is executed when the intrinsics are referenced.

The algorithms which are employed by the system for mathematical intrinsics are discussed in the Mathematical Intrinsics Information Manual, form number 5000151.

An intrinsic is referenced in the same manner as a function subprogram.

The intrinsics CDABS, CDCOS, CDEXP, CDLOG, CDSIN, CDSQRT, DCMPLX, and DCONJG are identical to the intrinsics CABS, CCOS, CEXP, CLOG, CSIN, CSQRT, CMPLX, and CONJG, respectively.

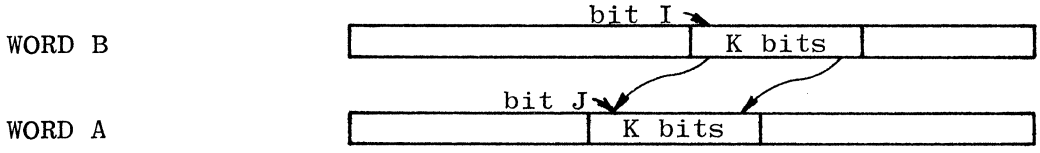
Intrinsics whose types do not agree with the types which would be assigned to their names by default need not be named in type statements in the program units in which they are referenced. Intrinsics differ from other functions in this respect.

- **CONCAT Intrinsic**

The CONCAT intrinsic is provided to allow partial word manipulation whereby one field of a data word may be placed into a selected field of another data word.

This intrinsic takes five actual arguments. The first two of these arguments must be integer or real variables. The last three arguments must be non-negative arithmetic expressions. Thus a typical CONCAT reference has the form: CONCAT(A,B,I,J,K). The truncated values of I and J must be less than 48; the truncated value of K must be less than or equal to 48.

When CONCAT is referenced, K bits are taken from word B (where J is the number of the left-most of these bits in B) and placed in word A in such a manner that I is the number of the position of the left-most bit after transfer. Word A is the value resulting from the intrinsic reference. (Bit 47 is the left-most bit of a data word.) Thus, the value of the field [J:K] in B is transferred to the field [I:K] in A. (See chapter 11.)



The following is an example of a program unit referencing the CONCAT intrinsic:

```
FUNCTION UNITE(P1,P2)
UNITE=CONCAT(P1,P2,23,47,24)
RETURN
END
```

The value returned when this UNITE function is referenced is a word constructed from P1 and P2 in such a manner that the left-most 24 bits of P1 and the left-most 24 bits of P2 are placed into the word in that order.

Another example using the CONCAT intrinsic appears in the section on FUNCTION subprograms.

Greater efficiency may be obtained by using the AND and OR intrinsic, whenever possible, instead of the CONCAT intrinsic.

- **CTIME Intrinsic**

Appearance of the special compile-time intrinsic CTIME in a source statement returns the contents of certain internal timing registers at the time the compiler encounters the construct. The CTIME intrinsic accepts the same range of parameter and returns values of the same type as the TIME intrinsic. The parameter must be an integer or real constant -- real parameters will be truncated to obtain an integer.

This intrinsic has effect only at compilation time. The compiler replaces the CTIME reference by the integer value contained in the appropriate timing locations at that point in time. For example, if the assignment statement:

```
X = CTIME(1)
```

appears in the source input, the compiler will substitute for CTIME(1) the current time of day which is an integer value denoting multiples of one-sixtieth of a second. When the indicated statement is executed, the variable X will be assigned this constant integer value.

A reference to CTIME may not appear in an initial value list.

- **RANDOM Intrinsic**

A reference of the RANDOM intrinsic returns a pseudo-random number between 0 and 1. The actual argument passed to this intrinsic must be an integer or real variable. The value of this variable is changed each time the intrinsic is executed; the value returned to the argument is also a pseudo-random number but is an integer in the interval 1 to 2**39-1. Thus the value of the argument should not be changed in the program after the first reference of RANDOM.

The value returned by the intrinsic is from a uniform distribution.

- **TIME Intrinsic**

A reference of the TIME intrinsic returns a value which represents the present contents of various internal timing registers. The argument must have a value when truncated in the interval 0 to 15. The following argument values return the indicated results:

0 - returns the current date in BCL-coded digits (in the format: 0yyddd, where y is a year digit and d is a day digit);

1 - returns an integer value which is the time of day in units of sixtieths of a second;

2 - returns an integer value which is the elapsed processor time of the job in units of sixtieths of a second;

3 - returns an integer value which is the elapsed I/O time of the job in units of sixtieths of a second;

4 - returns an integer value which is the current contents of a 6-bit machine clock which increments every sixtieth of a second;

5 - returns the same as 0, except that the date is expressed as: 00mmddyy, where m is a month digit;

6 through 9 - return an integer zero;

10 - returns the same as 0, except that the digits are EBCDIC-coded;

11 - returns the same as 1, except that the time unit is 2.4 microseconds;

12 - returns the same as 2, except that the time unit is 2.4 microseconds;

13 - returns the same as 3, except that the time unit is 2.4 microseconds;

14 - returns an integer value which is the current contents of a 36-bit machine clock which increments every 2.4 microseconds;

15 - returns the same as 5, except that the digits are EBCDIC-coded.

SUBPROGRAMS AND INTRINSICS

The following is a sample program unit employing the TIME intrinsic. The subroutine DATIT returns the current date in the form: mm/dd/yy left-justified in the array, DATE.

```
SUBROUTINE DATIT(DATE)
REAL DATE(2)
DATE(1)='  /  /'
DATE(2)='      '
B=TIME(15)
DATE(1)=CONCAT(DATE(1),B,47,47,16)
DATE(1)=CONCAT(DATE(1),B,23,31,16)
DATE(2)=CONCAT(DATE(2),B,47,15,16)
RETURN
END
```

CHAPTER 16

ADDITIONAL LANGUAGE CAPABILITIES

The B 6700 implementation of FORTRAN contains various powerful language abilities, some of which have already been discussed previously. These include partial word manipulation using the CONCAT intrinsic (chapter 15), allowing arbitrary expressions in the DO statement (chapter 8) and as subscripts (chapter 5), and allowing recursive subroutines (chapter 15). Two additional features are discussed in this chapter.

One of the B 6700/B 7700 FORTRAN language abilities consists of an expansion of the concept of subroutines whereby a FORTRAN program may employ subprogram units written in B 6700 ALGOL or COBOL. Thus, a CALL statement may reference a subroutine not written in FORTRAN. The subprogram units thus referenced must be separately-compiled and bound to the FORTRAN program in a prescribed manner.

Also, a language extension exists which consists of the ability to initiate independent programs from within an executing B 6700 FORTRAN program. The basis of this ability is the executable ZIP WITH statement.

SEPARATELY—COMPILED PROGRAM UNITS

The feature of B 6700/B 7700 FORTRAN to be dealt with herein is a Burroughs enhancement of the language abilities via system facilities and is discussed in detail in the B 6700 Program Binder Information Manual, Form No. 5000045. This feature is the result of the fact that the user may bind FORTRAN, ALGOL, and/or COBOL program units into a single program structure. In this way a B 6700/B 7700 FORTRAN program may make use of the features available to the B 6700/B 7700 ALGOL and COBOL languages.

The binding process involves the separate compilation to library of program units and their linking to the program via the program binder prior to execution. The present discussion will not attempt to explain the details of this

process, but it will give an example of the way in which the FORTRAN language may be used in connection with this system feature.

The process to be displayed consists of the separate compilation of main program and subroutine program units in various languages, the binding together of these units into a program structure, and the execution of this final program construct.

The following five program decks exemplify the use of the Program Binder to extend the FORTRAN language to make use of extended ALGOL constructs:

NOTE

<I> represents an invalid character in card column 1.

Deck 1:

```
<I>COMPILE FORT/HOST WITH FORTRAN LIBRARY; EBCDIC
      DIMENSION X(1), Y(1)
      X(1)='3ABCDE'
      Y(1)='MENTAT'
      WRITE(6,10)X,Y
 10   FORMAT(2(X,A6))
      CALL SUB(X,Y)
C***CALL EXCHANGE SUBROUTINE (FORTRAN)
      WRITE(6,10)X,Y
      CALL SUBA(X,Y)
C***CALL PARTIAL EXCHANGE PROCEDURE (ALGOL)
      WRITE(6,10)X,Y
      STOP
      END
<I>END
```

Deck 2:

```
<I>COMPILE FTEST/FORT WITH FORTRAN LIBRARY; EBCDIC
 $SET SEPARATE
       SUBROUTINE SUB(A,B)
       DIMENSION A(1),B(1)
 C***SUBROUTINE EXCHANGING A AND B VALUES
       C=A(1)
       A(1) = B(1)
       B(1)=C
       RETURN
       END
<I>END
```

Deck 3:

```
<I>COMPILE FTEST/ALG WITH ALGOL LIBRARY; EBCDIC
   PROCEDURE SUBA(A,B); ARRAY A,B[*];
   %  SUBROUTINE REPLACING FIRST CHARS. IN A
   %  BY (1ST CHAR. IN B) CHARACTERS IN B
   BEGIN
      POINTER P,Q;
      LABEL XIT;
      P:=POINTER(A); Q:=POINTER(B);
      IF Q LSS "0" OR Q GTR "5" THEN GO XIT
      ELSE REPLACE P BY Q+1 FOR N INTEGER(Q,1);
   XIT: END.
<I>END
```

Deck 4:

```
<I>BIND BOUND/PROGM WITH BINDER FOR LIBRARY
<I>BINDER FILE HOST(TITLE=FORT/HOST)
<I>EBCDIC
 $SET LIST
 BIND = FROM FTEST/=;
<I>END
```

ADDITIONAL LANGUAGE CAPABILITIES

Deck 5:

```
<I>RUN BOUND/PROGM
<I>END
```

Resultant Output:

```
3ABCDE MENTAT
MENTAT 3ABCDE
ABCTAT 3ABCDE
```

The example just displayed exhibits the binding of a FORTRAN subroutine and an ALGOL procedure to a FORTRAN main program. The FORTRAN subroutine performs an argument exchange, while the ALGOL procedure performs a partial exchange using pointer expressions. Thus, the main program is able to employ the powerful B 6700/B 7700 ALGOL pointer facility by merely calling a bound-in procedure.

INDEPENDENT PROCESS INITIATION

Another feature of B 6700/B 7700 FORTRAN is the ability of an executing program to initiate independent programs. The FORTRAN construct involved is the B 6700/B 7700 extension to the language consisting of the executable ZIP WITH statement. The purpose of this statement is to accomplish the program initiation at a specific point within a FORTRAN program.

The ZIP WITH statement will be discussed here in detail since it is a unique Burroughs FORTRAN extension.

The proper format of the ZIP WITH statement is:

ZIP WITH a or ZIP WITH e
where e is an arithmetic expression, the value of which is a file designator and a is an array name.

The ZIP WITH statement causes the MCP to activate a library program using as control and program parameter card images the information contained in the array indicated by the array identifier or the file referred to by the file designator. (The file is designated by the integer portion of the result of the evaluation of the <arithmetic expression>.) Thus, a program may be initiated completely independent of the initiating program. (For information concerning system control cards, see B 6700/B 7700 System Software Handbook, Form No. 5000722.) Specific rules govern both forms of the ZIP WITH statement. The contents of both the array and the designated file specify the process to be initiated and must follow a prescribed format. The appropriate formats will be discussed here as well as the action resulting when the ZIP WITH statement is executed.

Zip With Array

A ZIP WITH statement may be executed with an array identifier as its argument. This array must contain control statement information consisting of EBCDIC characters appearing as it might normally appear on punched cards, i.e., as strings of characters. The first character of the first word of this array may be a blank or a "?" and initiates the control statement information.

The subsequent control statements must be separated by single semicolons (;) and must be terminated by a semicolon followed by the word "END" and a period (.), in that order. The array involved is processed as one punched card, but it may include more than 72 characters. Each array element contains six characters (or 12 characters in the case of double precision arrays), and the array must be dimensioned large enough to hold the desired control statement information.

When the ZIP WITH statement is executed, the MCP examines the contents of the array for validity and prints a message on the operator's console if any errors are detected in this information. If no errors are detected, the control statement information is obeyed. After the execution of the ZIP WITH statement, program control passes to the next statement in sequence, with any programs initiated by the ZIP WITH statement executing independently.

ADDITIONAL LANGUAGE CAPABILITIES

An example of a FORTRAN program using the ZIP WITH statement with array arguments is presented here and explained in later paragraphs in this chapter:

```
      DIMENSION B(3), C(4), D(2,2)
      DATA B/'?RUN ZIP/TEST;END.'/
 1    WRITE(6,2)B
 2    FORMAT(X,3A6)
 3    ZIP WITH B
      C(1)='?RUN M'
      C(2)='AR/VEL'
      C(3)='(6);  '
      C(4)='END.  '
 4    WRITE(6,5) C
 5    FORMAT(X,4A6)
 8    ZIP WITH C
      D(1,1)=' RUN P'
      D(2,1)='ROGRAM'
      D(1,2)='/ONE;'
      D(2,2)='END.  '
 9    WRITE(6,5) ((D(I,J),I=1,2),J=1,2)
10    ZIP WITH D
      STOP
      END
```

The sample program just displayed contains three examples of the use of the ZIP WITH statement utilizing three arrays. The three examples consist of the insertion of control statement information into arrays which are then used as ZIP WITH statement arguments.

In the first example the control statements: "?RUN ZIP/TEST;END." are placed via a DATA statement into array B which has been properly dimensioned to hold this information. Thus, when statement number 1 is executed, the printed output is:

```
?RUN ZIP/TEST;END.
```

When the statement with the label 3 is executed, the library program ZIP/TEST is initiated.

In the second example, continuing the program after statement 3, array C is filled with the control statements: "?RUN MAR/VEL(6); END.". Therefore, the printed output resulting from the execution of statement 4 is:

```
?RUN MAR/VEL(6);  END.
```

When statement number 8 is executed, the library program MAR/VEL is initiated with the literal 6 passed as a parameter. The "?" in the present example was unnecessary for the successful execution of the ZIP WITH statement, but it was acceptable and could be useful in identifying arrays containing control statement information within printed output. (Only one such question mark may appear in an array.) Any other character (except a blank) placed into the leftmost character position of C(1) would alter the intent behind the ZIP WITH C statement. Thus, the execution of MAR/VEL would not have been initiated if, for example, C(1) had contained "XRUN M" when statement 8 was executed.

In the third example, concluding the program after statement 8, the two-dimensional array D is filled with the control statement: " RUN PROGRAM/ONE; END. ". Since a two-dimensional FORTRAN array such as this is stored internally by columns as a one-dimensional array, the control statement information must be stored sequentially in the array elements in such a way as to position the information correctly in the corresponding one-dimensional internal array. Hence, in this example the information is stored in successive elements down each array column sequentially, beginning with the first column. The array elements are stored internally in the same order as that in which statement number 9 outputs the D array elements. When this WRITE statement is executed, the printed output is:

```
RUN PROGRAM/ONE; END.
```

When statement number 10 is executed, the library program PROGRAM/ONE is initiated.

ADDITIONAL LANGUAGE CAPABILITIES

Zip With File

A ZIP WITH statement may be executed with a disk file designator as its argument. The file designator must be an arithmetic expression corresponding to an internal file name. The designated file, in turn, must contain control, program, and data information appearing as it might normally appear on punched cards, i.e., as strings of characters. The first character in this file must be a "?"; this character initiates the control statement information.

The subsequent control statements may be separated by question marks or semicolons, but the file must adhere to pseudo reader formats (see below). All control card images should be EBCDIC records complying with standard control card syntax. If any records in addition to these within the file are BCL, all records following the ?BCL record should be BCL coded up to and including the ?END card image record or the next control card image for stacked decks. Following this latter card, all subsequent control cards must again be EBCDIC, etc., for all stacked decks. Thus, program data may be interspersed among the control statement information as in ordinary program card decks.

File Format

The files under consideration should also agree with pseudo readers as to record size and disk area (row) size. Presently, all such file records must be 15 words long and blocked so that there are two records per disk segment. The actual block size is otherwise unimportant except that it must have an area (row) size which is an integer multiple of seven disk segments (i.e., of 14 logical records or 210 words).

When such a file is used as the argument of a ZIP WITH statement and that statement is executed, the file is passed to the MCP and the program then continues processing in sequence. The argument of the ZIP WITH statement must be a valid FORTRAN file designator, i.e., an arithmetic expression. If the referenced file is not a disk file or if it is declared to be a disk file but is not currently on disk, then the ZIP WITH statement is effectively ignored. The contents of the file thus referenced, if found, are examined by the MCP for validity and the control statement information contained therein is obeyed if no errors are found. The file is then removed from the system.

Example

An example of the creation of valid files and the use of such files as ZIP WITH statement arguments is given here. A possible method of creating such files is displayed as is an example of the use of these files as ZIP WITH statement arguments. Thus, two FORTRAN programs are examined in the following paragraphs.

- Constructing the File

The execution of the following program constructs two disk files, ZIPDISK/ONE and ZIPDISK/TWO, which are suitable for use as ZIP WITH statement arguments in later FORTRAN programs:

```
      FILE  10=ZIPDISK/ONE, UNIT=DISK, RECORD=15,BLOCKING=2,
     1                      AREA=14*1,SAVE=3
      FILE  11=ZIPDISK/TWO, UNIT=DISK,RECORD=15,BLOCKING=2,
     1                      AREA=14*1,SAVE=3
            DIMENSION B(6),D(15)
            DATA BLANK/Z404040404040/
            B(1)='?RUN P'
            B(2)='ROGRAM'
            B(3)='/ONE;O'
            B(4)='PTION='
            B(5)='DSED; '
            B(6)='END.  '
     1      WRITE(6,2) B
     2      FORMAT(X,6A6)
     3      WRITE(10,4) B
     4      FORMAT(6A6)
     5      LOCK 10
            D(1)='?RUN J'
            D(2)='OB/SIX'
            D(3)=';DATA '
            D(4)='FILE5 '
            DO 6 I=5,15
     6      D(I)=BLANK
     7      WRITE(6,8)D
     8      FORMAT(X,15A6)
     9      WRITE(11,10) D
     10     FORMAT(15A6)
            C=1.2
     11     WRITE(6,13) C
     12     WRITE(11,13) C
     13     FORMAT(F4.1)
            D(1)='?END. '
     14     WRITE(6,8) D
     15     WRITE(11,10) D
     16     LOCK 11
            STOP
            END
```

The program just listed provides two examples of the way in which one may construct disk files suitable for use as ZIP WITH statement arguments. One of the files so constructed contains only control statement information, while the other file contains program data in addition to control information.

To begin with, the program uses two file declaration statements to declare the files to be constructed. (See appendix B.) The first file is given an internal file name of FILE10 (hence the file designator 10 in the first file declaration statement) and a TITLE (file directory name) of ZIPDISK/ONE. The second file declared is given an internal name of FILE11 (hence the file designator 11 in the corresponding file declaration statement) and a TITLE of ZIPDISK/TWO. The UNIT, RECORD, BLOCKING, AREA, and SAVE clauses of both file statements declare both files to be assigned to disk, have records 15 words in size, be blocked two records per disk segment, have an area (row) size of 210 words (14 records or seven disk segments), and be saved on disk for three days from the date creation.

Next, the control statement information: "?RUN PROGRAM/ONE; OPTION=DSED;END." is placed into the first record of file 10 (the file with the internal name of FILE10). When statement number 1 is executed, the printed output will duplicate the equivalent contents of the first six words of this file. Since this is the only information to be placed into this file, the END statement is followed by a period. This period will terminate the MCP scan of this information after being initiated by a ZIP WITH statement referencing this file, so the contents of the other 204 words of this file may be neglected. Statement number 3 places the desired information within the file, and the LOCK statement (statement number 5) secures the file within the directory to be accessed in later programs.

File 11 (the file with the internal name of FILE11) is then filled with the appropriate information. This information consists of both control statement information and data of the form:

```
?RUN JOB/SIX;DATA FILE5
 1.2
?END.JOB/SIX;DATA FILE5
```

where each line represents one file record (also as printed by statement numbers 7, 11, and 14) and the MCP scan is terminated by the period following the END statement. The information in the third record following the period is ignored, as is all information in the following 11 records of this file.

Statement numbers 9, 12, and 15 place the desired information within the file, and the LOCK statement (statement number 16) secures the file within the directory (with the TITLE ZIPDISK/TWO) to be accessed in later programs.

- Use of the File

An example of a program accessing this file is given here. The following deck is an example of a B 6700/B 7700 FORTRAN program deck employing two ZIP WITH statements with file arguments. This deck will be discussed in the following paragraphs:

```
<I>COMPILE ZIP/RUN WITH FORTRAN
<I>FILE FILE4(TITLE=ZIPDISK/ONE,KIND=DISK)
<I>FILE FILE8(TITLE=ZIPDISK/TWO,KIND=DISK)
<I>EBCDIC
   1   ZIP WITH 4
   2   ZIP WITH 8
       STOP
       END
<I>END
```

The deck just listed provides an example of a FORTRAN job using the file ZIP statement. Two file control cards are employed to equate the internal file names FILE4 and FILE8 to the external file names (i.e., TITLEs) ZIPDISK/ONE and ZIPDISK/TWO, respectively. These two files are then used as arguments in ZIP statements.

When the two ZIP WITH statements are executed, the contents of the two files are passed to the MCP for processing and the files are removed from the system. The execution of statement number 1 results in the execution of the library program entitled PROGRAM/ONE with the option DSED set. The execution of statement 2 results in the execution of the library program entitled JOB/SIX with the "1.2" data record prepared to be read from a file with a TITLE of FILE5 by PROGRAM/ONE with a READ statement employing an F4.1 editing specification. These two initiated jobs are free to run independently of the present program. The initiating program (the job containing the two ZIP WITH statements) is free to reach EOJ without having an effect on the execution of PROGRAM/ONE or JOB/SIX.

CHAPTER 17

DEBUGGING AND TIMING AIDS

INTRODUCTION

DEBUG statements assist in isolating programming errors and in identifying heavily executed program sections so that more efficient programming techniques may be applied to these sections. All such statements contain the characters DEBUG in columns 1 through 5 and conform to the usual FORTRAN statement formulation rules. Once a program has been properly corrected, the output from all DEBUG statements may be deleted by compiling with the compiler option OMITDEBUG set. Thus, DEBUG statements can become a permanent part of a symbolic program, included while the program is being checked out and excluded while the program is in production form.

DEBUG MONITOR STATEMENT

The DEBUG MONITOR statement provides the ability to monitor the changes in value of variables and/or array elements within a program unit. The proper syntax for the DEBUG MONITOR statement is:

DEBUG MONITOR (n) m
where DEBUG appears in columns 1 through 5; MONITOR begins in or after column 7; n is a constant file designator; and m is a list of variables and/or array elements separated by commas.

The file designator must be an integer literal (1-99) that specifies the file to which the monitoring output is to be directed.

The Monitor List may be a series of variables and/or array identifiers separated by commas. If the list contains an array identifier without subscripts, then every element of that array is monitored. If the array identifier is followed by a series of subscripts enclosed in parentheses and separated by commas, then only those elements in the subscript list are

monitored. An individual subscript is either a constant or a single variable. It may not be an arithmetic expression. Only the first appearance of a simple variable or an array identifier in a monitor list is processed. All successive appearances are ignored, and a warning message is issued.

At execution time, whenever the value of a monitored variable is changed, diagnostic information, including the old and new values of the variable, is written on the designated file. Monitoring action takes place for a simple variable whose value is changed through an assignment statement, a READ statement, or through being passed as an actual parameter to a subprogram. Monitoring action occurs for an array element only when its value is changed through an assignment statement.

There can be more than one DEBUG MONITOR statement per program unit, and DEBUG MONITOR statements can be continued through continuation cards. Arrays must be dimensioned prior to appearing in a DEBUG MONITOR statement.

Example

```
      DIMENSION A(5), B(5), C(5)
DEBUG MONITOR (6) X, A(1), (I), (5), B, Y, C, Z, A
      I=3
      A(3)=1
      B(4)=2
      X=3
      A(4)=4
      READ (5,/) Y, A(1)
      CALL SUB(Z,C)
      STOP
      END
      SUBROUTINE SUB(T,V)
      DIMENSION V(5)
      V(1)=4
      T=3
      RETURN
      END
```

With input data (5,5), the monitoring output for this example will be:

```
A(3)=0=1
B(4)=0=2
X=0=3
Y=0=5
Z=0=3
```

DEBUG DUMP STATEMENT

The DEBUG DUMP statement provides the ability to monitor the value of a list of variables and/or arrays within a program unit when program control passes to a label. The proper syntax for the DEBUG DUMP statement is:

1. DEBUG DUMP(n) s,f(j=k,m)/d 2. DEBUG DUMP(n) s,f/d 3. DEBUG DUMP(n) s(j=k,m)/d 4. DEBUG DUMP(n) s/d
where DEBUG appears in columns 1 through 5; DUMP begins in or after column 7; n is a constant file designator; s is a program label; f is a positive integer denoting frequency; (j=k,m) is a control condition with j the control variable and k and m positive integers denoting lower and upper bounds; and d is a list of variables and/or array names separated by commas.

The file designator, n, is an integer literal (1-99) that specifies the file to which the diagnostic information is to be directed.

When program control passes to the specified label, dump action depends in part on the frequency f; the diagnostic output will occur every fth time program control passes to the label. If frequency, f, is not specified, it will have a default value of 1.

The control condition also sets restrictions on when dump action will occur by designating a program control variable j with upper and lower bounds k and m. As long as j is bounded by k and m, then diagnostic output will be produced upon transfer to label s, subject also to the frequency f. If the lower bound k is not specified, it will have a default value of 1. If upper bound m is not specified, there will be no upper bound on the control variable j. If a control condition is not specified, dump action will be dependent only on the frequency f and the number of times execution control passes to label s.

DEBUGGING AND TIMING AIDS

The DUMP list is a series of variables and/or array names separated by commas. Specific array elements may not be specified in the list.

There may be more than one DEBUG DUMP statement per program unit, and DEBUG DUMP statements may be continued through continuation cards. Each such statement must follow the dimensioning of all arrays and precede the occurrence of all labels referenced by it.

Example

```
DEBUG DUMP (6) 10,2 (K=1,15)/M
      DO 10 M=1,6
      K=M+10
   10 CONTINUE
      STOP
      END
```

The following is the diagnostic output for this example:

```
10 FROM 0002:000B:0
M=2
10 FROM 0002:000B:0
M=4
```

DEBUG PROGRAMDUMP STATEMENT

The DEBUG PROGRAMDUMP statement provides a "snapshot" of the program stack by invoking the MCP procedure PROGRAMDUMP at the point in the program where the statement is inserted. The proper syntax for the DEBUG PROGRAMDUMP statement is:

DEBUG PROGRAMDUMP
where DEBUG appears in columns 1 through 5; and PROGRAMDUMP begins in or after column 7.

The DEBUG PROGRAMDUMP statement causes the program execution to be temporarily suspended while the contents of the program's stack are dumped. The program will then be resumed without the dump affecting the program's execution. The options available to the MCP PROGRAMDUMP procedure (i.e., arrays, code, and files) can be invoked by using an OPTION control card specifying the desired options.

DEBUG STATISTICS STATEMENT

The DEBUG STATISTICS statement produces a frequency/timing study of a program's execution. Heavily executed areas of a program can thus be isolated and perhaps improved through more efficient programming. The proper syntax for the DEBUG STATISTICS statement is:

1. DEBUG STATISTICS(m) 2. DEBUG STATISTICS(m) BLOCKS 3. DEBUG STATISTICS DIAGNOSTICS 4. DEBUG STATISTICS DIAGNOSTICS BLOCKS 5. DEBUG STATISTICS NO
where DEBUG appears in columns 1 through 5; STATISTICS begins in or after columns 7; and m is a constant file designator.

The file designator m, must be an integer literal (1-99) that specifies the file to which the statistics output is to be directed. DIAGNOSTICS may be substituted for the constant file designator to indicate that the system diagnostic file is to be used for the STATISTICS output.

Two options are available: subprogram times and counts only, or times and counts for each logical segment or block in each subprogram. The first of these options is specified by leaving the columns following the file designator blank, while the second is specified by following the file designator with BLOCKS. The statement DEBUG STATISTICS NO resets STATISTICS.

Any time the program executes a STOP or DUMP STATISTICS statement or the program's exceptionevent (<mixid>HI) is caused, the accumulated statistics are output to the indicated file. The subprogram and block (if applicable) being executed at the time the statistics output occurs are flagged with an asterisk to indicate that the elapsed time for this program and block includes the time necessary to output the statistics, and thus may be slightly inaccurate.

When the BLOCKS option is specified, program execution time increases substantially (30 to 50 percent). The DEBUG STATISTICS statement stays in effect until reset.

DEBUGGING AND TIMING AIDS

DEBUG DUMP STATISTICS STATEMENT

The DEBUG DUMP STATISTICS statement causes the accumulated timing/frequency statistics to be output (see DEBUG STATISTICS statement in this chapter). The proper syntax for the DEBUG DUMP STATISTICS statement is:

DEBUG DUMP STATISTICS
where DEBUG must appear in columns 1 through 5; and DUMP must begin in or after column 7.

DEBUG TRACE STATEMENT

The DEBUG TRACE statement traces a program's flow by printing out all sequence numbers and labels of specified statements and subprogram calls in execution order. Comparator action may also be specified to detect deviations in predetermined program flow and to have program dumps taken if a mismatch should occur. The proper syntax for the DEBUG TRACE statement is:

1. DEBUG TRACE (f,c)t 2. DEBUG TRACE (f)t 3. DEBUG TRACE t
where DEBUG appears in columns 1 through 5; TRACE begins in or after column 7; f and c are constant file designators; and t is a list of TRACE options separated by commas.

The file designators, f and c, must be integer literals (1-99) where f is the output file and c is the comparator file. The value of c may specify either a disk or tape file. If f is not specified, then the previous value of f is assumed. If f has never been specified, then tracing action is suppressed. If c is not specified, then the trace output is written on the file f. If c is specified then f is not used for the trace output. Instead, a record is read from file c and compared to the trace record that would have been output to f, if c had not been specified. If the records compare as identical, then no action is taken. If they do not compare then a message is output to file f; and if DUMP (or PROGRAMDUMP) is specified in the Trace List, then a program dump is taken.

Trace List

Specifications in the Trace List may be DUMP, NODUMP, PROGRAMDUMP, NOPROGRAMDUMP, CALLS, NONE, S, and #S where S is defined to be any of the following: ALL, IO, GO, GO TO, READ, WRITE, BACKSPACE, REWIND, PRINT, PUNCH, =, CALL, and CONTINUE. DUMP specifies that a program dump is to occur if the record comparison with file c fails. PROGRAMDUMP is equivalent to DUMP. NODUMP or NOPROGRAMDUMP resets this option. None of these options have any meaning if the comparator mode has not been requested (that is, if file c has not been specified). CALLS traces all subroutine and function entries and exits; however, intrinsics calls are not traced. NONE resets all requests preceding it. Otherwise, each trace specification is processed accumulatively.

The specification S is used to indicate that tracing is desired on an executable statement or on a class of executable statements. If proceeded by a pound sign (#) then only labelled statements of the specified class are traced. An explanation of the various statement classes follows:

ALL	indicates all executable statements including subprogram entries and exits.
IO	indicates all input/output statements.
GO, GO TO	indicates all input/output statements.
PRINT, PUNCH, READ, WRITE, CALL, CONTINUE, BACKSPACE, REWIND	indicate the like-named statements.
=	indicates all assignment and DO statements.

CHAPTER 18

FORTRAN COMPILER

INTRODUCTION

The purpose of the present chapter is to provide an outline of the features of the B 6700/B 7700 FORTRAN compiler. An installation-directed description of the compiler and a digest of user-oriented compiler information are presented The programmer is referred to the user-oriented section for information concerning the submitting of FORTRAN programs to this compiler as well as a discussion of the compiler's capabilities.

The B 6700/B 7700 FORTRAN compiler is a special-purpose program written in B 6700/B 7700 Extended ALGOL which compiles to yield an object program designed to accept as input source programs written in the FORTRAN language described in the preceding sections, producing object programs from these input programs. The FORTRAN compiler described here and the object programs generated by this compiler are designed to operate under control of the B 6700/B 7700 Master Control Program (MCP).

INSTALLATION/COMPILER INTERFACE

The B 6700/B 7700 FORTRAN compiler has certain minimum system requirements which must be satisfied before it may function, and it is maintained in the same manner as other standard B 6700/B 7700 software items. These requirements and maintenance procedures are discussed here.

System Requirements

In order to operate, this compiler requires a minimum hardware and software system environment as described here.

Required Hardware

Before this FORTRAN compiler can operate, the following minimum hardware configuration must be provided:

a. One card reader.
b. One line printer (with a minimum of 120 print positions per line of printed text).

c. Two memory modules (i.e., 32,768 52-bit words of storage).

d. Approximately 30,000 words of disk storage (to hold the compiler).

The addition of hardware to this minimum system will result in an increase in the capabilities of the compiler. The addition of one magnetic tape unit will allow source input from magnetic tape or new source output on magnetic tape. The addition of two magnetic tape units will allow both source input and new source output on magnetic tape. Additional memory modules may result in a faster compilation rate, and additional disk capacity may allow the compilation of larger input programs.

Required Software

The FORTRAN compiler must run under the control of the B 6700/B 7700 MCP. In addition, for compiler operation, an intrinsic file (e.g., SYSTEM/INTRINSICS) should also be on the system and loaded.

Maintenance of Compiler Software

The B 6700/B 7700 FORTRAN compiler, as a standard software item, can be copied (from the object file) or can be copied and compiled (from the source file), just as any user program can be copied, or copied and compiled, from a user's object or source file.

This compiler may be compiled by first copying the source file to disk and then using a standard Extended ALGOL compilation deck. The source file for the compiler has the title SYMBOL/FORTRAN. This item may be patched in the same manner as any program may be patched.

The following is a sample card deck which displays the manner in which the FORTRAN compiler may be compiled:

```
<I> COMPILE SYSTEM/FORTRAN WITH ALGOL FOR LIBRARY
<I> ALGOL FILE TAPE (TITLE=SYMBOL/FORTRAN)
<I> EBCDIC
$LIST MERGE SEQERR
        patch cards (if any)
<I> END
```

NOTE

An <I> in this deck denotes an invalid character in card column 1.

If the FORTRAN compiler is compiled with the following compiler control card in the compilation card input file:

```
$LIST MERGE SEQERR MODELII
```

the resultant compiler will generate object programs which will take advantage of the presence of a B 6700 Model II processor.

If several patches are to be made to this software item, the utility patch merge program (SYSTEM/PATCH) may be used. This utility program merges individual patches into a master patch deck. (Refer to B 6700/B 7700 System Software Handbook (500722) for more information concerning this program.)

USER/COMPILER INTERFACE

The purpose of the B 6700/B 7700 FORTRAN compiler is to accept application programs written in the B 6700/B 7700 FORTRAN language and to produce from these programs object code which may be executed on the B 6700/B 7700 system.

All object programs generated by this compiler conform in the following manner:

a. The layout of the object program is standard to the B 6700/B 7700 system.
b. Standard loading format is produced for each program compiled.
c. Standard linkage code to the MCP is generated for each program compiled.

Concurrent with the production of this program object code, the user is provided with compile-time debugging and diagnostic facilities and the ability to a limited extent to control the function performed by the compiler (such as in the area of compiler file handling). The latter ability is available via the FORTRAN compiler control statements.

The debugging and diagnostic facilities provided by this compiler are compile-time additions to the compiler-provided printer listing of input source statements. The following items are provided as diagnostic aids by the compiler:

a. Syntax-error numbers and messages, which are placed on the printer listing following the line of text bearing the incorrect statement.
b. Messages and numbers denoting warning errors, optionally placed on the printer listing following the line bearing the incorrect statement.
c. Various compiler information messages.
d. A complete listing of compiler-generated code, optionally placed on the printer listing.

Appendix C provides a list of error messages, plus the compiler procedures that may produce each message.

All user communication with the compiler and all compiler output are handled via compiler files. A discussion of the interface between the user and the FORTRAN compiler is therefore necessarily an examination of the features of these compiler files. The system of compiler files is illustrated in figure 18-1.

The following sections discuss:

- The configuration of the FORTRAN compiler files.
- The manner in which card input to the compiler is handled.
- The compiler control statements which may be included in the source input files.
- The compatability between the FORTRAN language acceptable as source input to this compiler and other FORTRAN languages.

Compiler Files

Compiler communication is handled through various input and output files. These compiler files are described in the following paragraphs.

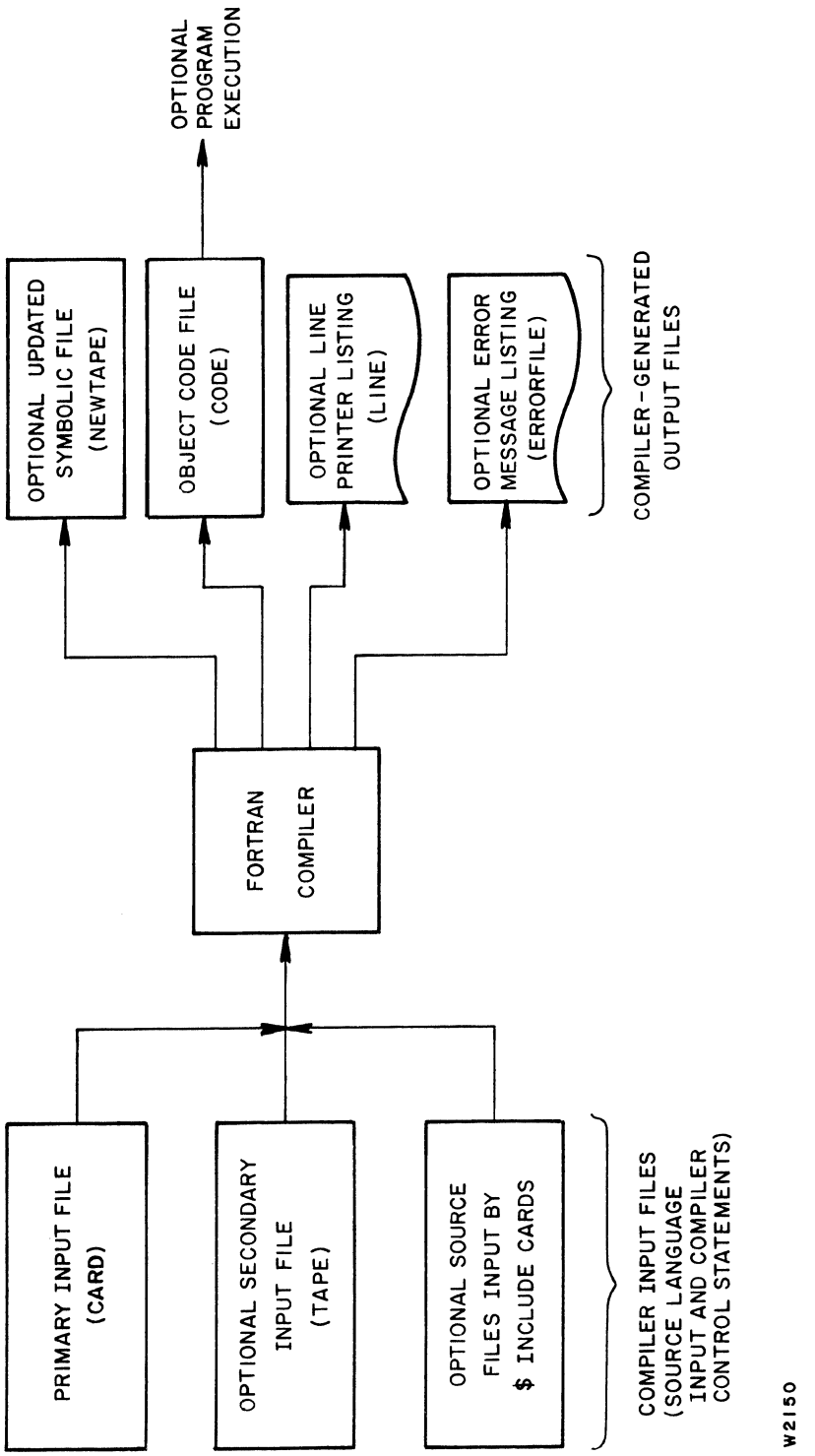

OPTIONAL PROGRAM EXECUTION
OPTIONAL UPDATED SYMBOLIC FILE (NEWTAPE)
OBJECT CODE FILE (CODE)
OPTIONAL LINE PRINTER LISTING (LINE)
OPTIONAL ERROR MESSAGE LISTING (ERRORFILE)
COMPILER-GENERATED OUTPUT FILES
FORTRAN COMPILER
PRIMARY INPUT FILE (CARD)
OPTIONAL SECONDARY INPUT FILE (TAPE)
OPTIONAL SOURCE FILES INPUT BY $ INCLUDE CARDS
COMPILER INPUT FILES (SOURCE LANGUAGE INPUT AND COMPILER CONTROL STATEMENTS)
W2150

Figure 18-1. FORTRAN Compilation System

FORTRAN COMPILER

Punched cards, disk, or magnetic tape may be specified as source language input medium for this compiler. Input must be in the input format defined in the preceding chapters of the present document. The compiler has the capability of merging, on the basis of sequence numbers, input from cards, tape, or disk. When inputs are being merged, indications of text insertions or replacements can be made to appear on the output listing. In addition to the output listing, the FORTRAN compiler can also generate updated symbolic output files. These files may be created in addition to the compiler-generated output code file. Compiler input and output files are discussed in detail here.

Input Files

The primary compiler input file is a card file with the internal name CARD; the secondary input file is a serial disk file with the internal name TAPE. The presence of the primary file (CARD) is required for each compilation; the presence of the secondary file (TAPE) is optional for each compilation. File CARD can be either BCL- or BCD-coded with 10-word records or EBCDIC-coded with 14-word records and may be either blocked or unblocked. File TAPE can be either BCL-coded with 10-word records or EBCDIC-coded with 14-word records and may be either blocked or unblocked. Both the CARD file and the TAPE file may be label-equated (via the FILE system control card) to change the file's TITLE and KIND. The file TAPE is used as input only when the MERGE compiler option is set.

Additional files may be input to this compiler through use of the INCLUDE compiler control statement. (See chapter 19, COMPILER CONTROL STATEMENTS.) These files may be either BCL-coded with 10-word records or EBCDIC-coded with 14-word records (similar to the TAPE file). These INCLUDEd files are by default disk files, and their internal names or TITLEs are as specified on the INCLUDE compiler control cards.

The FORTRAN compiler input files are listed in table 18-1.

Output Files

Output files produced by this compiler include the object code file, an updated symbolic file, a line printer listing, and an error message file. The object code file has the internal name CODE and is saved on disk after the compilation unless the COMPILE system control card specifies compilation for syntax only or unless syntax errors are detected in the source language input by the compiler. (See SAMPLE CARD INPUT DECKS, this chapter.) If compile-and-go is specified by the COMPILE card, then the object file is discarded after the code is executed. If compilation for library is specified, then the object

code file is saved on disk with an assigned SAVEFACTOR of 30 days. The TITLE of the saved code file is identical to the program name appearing on the COMPILE card except in the case of separately-compiled procedures. (See the Program Binder document.) When subprograms are compiled separately, the TITLE of the resultant object code library file consists of the program name appearing on the COMPILE card with the rightmost identifier in the program name replaced by the subprogram name. If there is only one identifier in the program name, this name is assigned as the code file TITLE.

The compiled program is logically segmented within the resultant code file by program unit. The code for each program unit begins at a physical disk segment boundary and fills as many disk segments as required within the limits of the system. An extremely long program unit may require more than one program segment, in which case it is automatically segmented as follows. (Such segmentation does not occur for separate compilations except for a main program.) As the amount of code generated for a program unit reaches 8192 words or 30*A words (where A is the numeric value of the AREASIZE of the code file), whichever is less, then program unit segmentation will occur. The code for the program unit is then contained in two program segments. This segmentation may occur more than once, depending on the size of the program unit.

The updated symbolic file is, by default, a disk file generated only if the compiler option NEW is set. This file contains the compilation source input or a selected portion of this input as specified by the states of the NEW and INCLNEW compiler options and may be used as the TAPE file or an INCLUDEd file for a succeeding compilation. This output file has the internal file name NEWTAPE and contains EBCDIC-coded 14-word records in 420-word blocks. The file is assigned a SAVEFACTOR of 10 days and a TITLE of FORSYM by the compiler.

The printer listing is an optional print file that is created unless the compiler option LIST is reset. (The LIST option is set by default unless the compilation is initiated through CANDE.) The file has the internal name LINE, contains 20-word EBCDIC-coded records, and bears the following information:

Compiler version.
Date and time of compilation.
Source and compiler control statements input to the compiler.
Code segmentation information next to the source input.
Error messages and error count.
Number of input card images scanned.
Elapsed compilation time.
Processing time required for the compilation.
Estimated size of the program's working stack when the program is executed.

Estimated size of the program files and related storage.
Total number of words of object code generated.
Number of words of array storage required for the execution of the program.
Number of segments in the generated program.
Number of disk segments required for the program code file.
Estimated core storage requirement.
The TITLE of the generated code file.
The date the compiler was compiled.

Depending upon the specified setting of various compiler options, this printer listing may contain more of fewer items than those listed above. Card images from file CARD are denoted on the listing by a "C" after the card contents. Card images from file TAPE are denoted by a "T" in this location. Card images from an INCLUDEd file are denoted by an integer from 1 to 5 indicating the level of INCLUDE nesting involved. A "P" denotes a CARD image that has superseded a TAPE card image.

The output error message file with the internal file name and assigned TITLE of ERRORFILE is an optional line printer file which is created when the ERRLIST compiler option is set. This file is normally employed for compilations initiated through CANDE, in which case ERRLIST is set by default and the ERRORFILE compiler file is assigned to the remote device involved. The file ERRORFILE may also be used for compilations initiated through the card reader. This file is assigned EBCDIC-coded 12-word records which results in a line width of 72 characters, allowing the file to be output to a remote terminal or card punch without truncation of text. When a syntax error is detected, the offending card image is written on this file, with an error message and indicator of the syntactical item in question being written on the following line of text.

The FORTRAN compiler output files are listed in table 18-1.

Compiler File Table

The FORTRAN input and output files are listed in the following table which provides information concerning the configuration of each file. This table lists the internal name of the file (i.e., the name used when the file is declared within the FORTRAN compiler), the purpose served by the file, the default KIND of the file, the code used to store file data, the default record size (MAXRECSIZE) and block size (BLOCKSIZE) of the file, and a brief commentary on the file.

The attributes of any of these files may be changed through use of FILE system control cards directed to the compiler.

Table 18-1. FORTRAN Compiler Files

INTERNAL NAME	PURPOSE	KIND	CODE	RECORD SIZE	BLOCK SIZE	COMMENTS
CARD	Input Card File	CARD READER (KIND = 9)	EBCDIC	14 Words	May be Blocked or Unblocked	Required for each compilation. Primary compiler input file; may be label-equated to change file attributes. CANDE file is equated to this file automatically for compilations initiated through CANDE. Default TITLE is CARD. BUFFERS = 5. FILETYPE = 7.
			BCL or BCD	10 Words		
TAPE	Input Disk File	DISK (KIND = 1)	EBCDIC	14 Words	May be Blocked or Unblocked	Optional file; need not be present for each compilation. Secondary compiler input file, selected as input by setting MERGE compiler option. May be label-equated to change file attributes as desired. The default TITLE is TAPE. FILETYPE = 7.
			BCL	10 Words		
May be Specified On INCLUDE Compiler Control Card	Library Include File	DISK (KIND = 1)	EBCDIC	14 Words	May be Blocked or Unblocked	Optional input file opened by $INCLUDE card bearing appropriate file TITLE or internal file name. Five levels (maximum) of nesting permitted. (See discussion of INCLUDE option.) FILETYPE = 7.
			BCL	10 Words		

Table 18-1. FORTRAN Compiler Files (Cont)

INTERNAL NAME	PURPOSE	KIND	CODE	RECORD SIZE	BLOCK SIZE	COMMENTS
CODE	Object Code File	DISK (KIND = 1)	Hexadecimal	30 Words	150 Words	Generated object code file. Saved or discarded and assigned a TITLE as indicated previously in text. If saved in the library, a SAVEFACTOR of 30 days is assigned. For CANDE compilations the TITLE becomes: OBJECT/ <file-title>. File TITLE during the compilation is DISKB/FORTRAN. AREASIZE = 180. BUFFERS = 2.
NEWTAPE	Updated Symbolic Output File	DISK (KIND = 1)	EBCDIC	14 Words	420 Words	Optional output file produced when NEW compiler option is set. This file contains portions of the source input, is label-equatable, is assigned a SAVE-FACTOR of 10 days and a default TITLE of FORSYM, and is suitable for use as a TAPE file for a later compilation. BUFFERS = 2. AREASIZE = 1000. AREAS = 60.

Table 18-1. FORTRAN Compiler Files (Cont)

INTERNAL NAME	PURPOSE	KIND	CODE	RECORD SIZE	BLOCK SIZE	COMMENTS
LINE	Line Printer Listing	LINE PRINTER OR DATACOM (KIND=39)	EBCDIC	20 Words	20 Words	Optional and label-equatable file produced when either the compiler option LIST or TIME is set. The default file TITLE is LINE.BUFFERS = 2.
ERRORFILE	Error Listing Output File	LINE PRINTER (KIND=7)	EBCDIC	12 Words	12 Words	Optional error listing file produced when ERRLIST compiler option is set. Contains card images and error messages. Automatically provided for CANDE input. The default file TITLE is ERRORFILE.

Sample Card Input Decks

The following discussion applies to FORTRAN compilations that do not utilize the batch facility discussed later in this chapter.

An input card file is required for every FORTRAN compilation. The format of this card file is discussed here.

For program compilation, the only required cards are (in proper order):

a. A COMPILE system control card to initiate the compilation and specify the type of compilation to be performed.
b. An EBCDIC or BCL system control card as a label on the source input card file.
c. Cards bearing the source program and compiler control cards (if any).
d. An END system control card.

In addition, MCP control cards may be placed after the COMPILE card to specify compilation parameters and to alter the attributes of compiler files. If a program execution is indicated on the COMPILE card, then MCP control cards may also be placed here to specify program execution parameters and to alter the attributes of program files (overriding FILE cards within the source deck). If the program execution requires input data, then the source cards may be immediately followed by an EBCDIC, BCL, or BINARY system control card (depending on the code employed on the data cards) bearing the TITLE of the program file and the cards bearing the program data. If this data is coded in BINARY format, then the entire deck is terminated by a binary end (BEND) card (as described in the B 6700 HANDBOOK in the discussion of "Control Statements and Control Cards"); otherwise, the standard END system control card is used.

For information concerning the initiation of FORTRAN compilations through CANDE, the user is referred to the Command and Edit (CANDE) Language Information Manual (5000318).

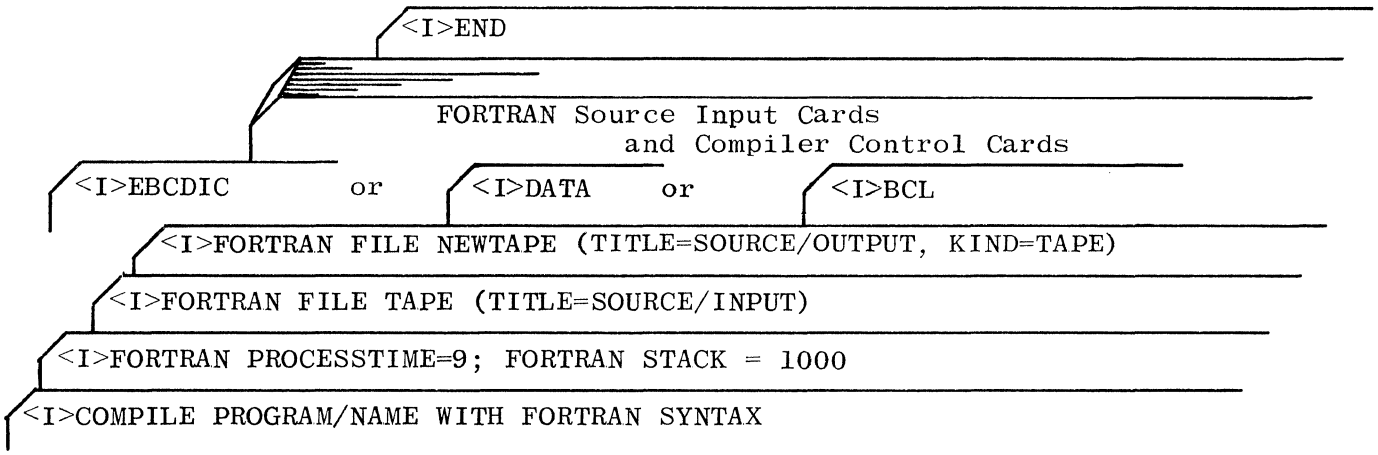
Four basic kinds of FORTRAN compilation are available to the B 6700/B 7700 user. They include:

- Compile for Syntax
- Compile for Library
- Compile and Go
- Compile for Library and Go

Examples of the proper format of compilation decks for each of these four kinds of compilation are provided here. In these decks the symbol <I> denotes an invalid character code punched in the first column of the card. (One example of such an invalid character code is a 1-2-3 punch.)

Compile for Syntax

The following sample deck displays the general deck format to be employed when a FORTRAN compilation is to be initiated for syntax-checking purposes. In this case the object code file is not retained after the compilation. A NEWTAPE file, however, may be generated and saved on the library by setting the NEW compiler option.



The first card of the compilation deck is the COMPILE system control card which bears the program name (in this case, PROGRAM/NAME) and a disposition of SYNTAX which indicates that a compilation for syntax check is to be initiated.

The second card of this compilation deck is an example of an MCP control card which specifies compilation parameters. The PROCESSTIME control statement indicates that a maximum of 9 seconds of processing time be expended for the present compilation. The STACK control statement indicates that a minimum of 1000 words be assigned in memory for the working stack for the compilation. This card is optional.

The next card is a FILE (label-equation) MCP control card which indicates that the compiler file with the internal name TAPE (i.e., the secondary compiler input file) has the TITLE of SOURCE/INPUT. The presence of a control card similar to this card generally is associated with the use of a MERGE compiler control statement in the input card file. The use of this card is optional.

The following sample card in this deck is a FILE system control card which specifies that the compiler file with the internal name NEWTAPE (i.e., the updated symbolic file) be assigned the TITLE of SOURCE/OUTPUT and be placed on magnetic tape. The presence of an MCP control card such as this in a compilation deck generally is associated with the use of the NEW compiler control statement within the input card file. This FILE card is optional.

FORTRAN COMPILER

The following card in this sample deck is either an EBCDIC or a BCL system control card, depending on the character code used on the following input cards (DATA is the same as EBCDIC). If BCD (i.e., Hollerith) coding is used on the input cards, then the BCL control card is the appropriate choice and the BCD compiler option must be set prior to the appearance of the first BCD-coded input card. The present MCP control card is used to label the following input card deck. No file TITLE follows the EBCDIC or BCL statement as the following cards are automatically assigned to the compiler as its input card file by the system.

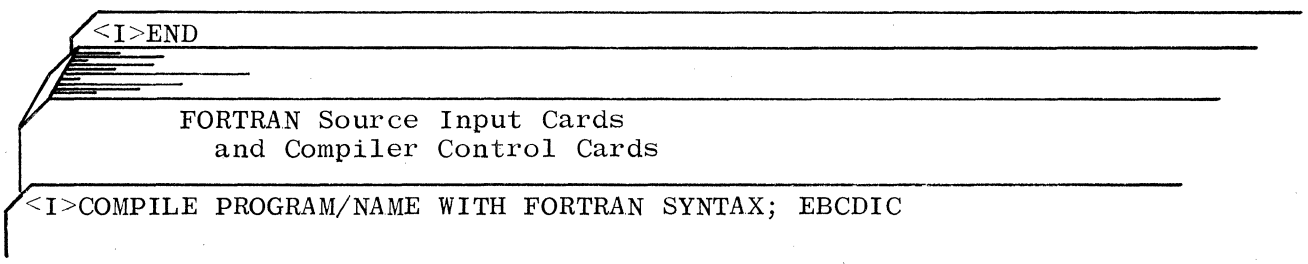
The next item in this sample compilation deck consists of the compiler input cards. These cards (up to the END system control card) are the records of the CARD compiler input file and bear FORTRAN program statements and compiler control statements. Each of the cards in this file may have eight-digit sequence numbers in card columns 73 through 80. These sequence numbers are used when input from this file and input from the TAPE file are merged (as specified by setting the MERGE compiler option). This card file need not be in the form of punched cards; instead, the compiler CARD file may be label-equated via a FILE system control card to another input medium (e.g., disk). In this case, the preceding EBCDIC or BCL card and the card input deck would be replaced by a FILE control card of the form:

```
<I>FORTRAN FILE CARD(TITLE=CDFILE,KIND=DISK)
```

This card specifies that (for this example) the card images which are records of the compiler input file CARD are contained in the serial disk file with the TITLE of CDFILE.

Each compilation deck must be terminated by a system control card, e.g., <I> END.

The general compilation deck described here may be reduced to a very simple form when the only input is a card file and all that is desired is a check of program syntax. This basic deck form is as follows (assuming EBCDIC-coded source input):



The minimal compilation deck requires:

a. A COMPILE system control card indicating the program name and that a FORTRAN compilation is to be initiated with the appropriate disposal (e.g., SYNTAX).
b. An EBCDIC or BCL system control card to label the input card deck.
c. The source input and compiler control cards (if any).
d. An END system control card to terminate the deck.

The first two items may be combined on a single card as shown in the previous example.

Compile for Library

A B 6700/B 7700 FORTRAN program may be compiled with the intention of retaining the generated code file among the library files. The basic difference between the general form of the compilation deck when syntax-checking only is desired (see "Compile for Syntax") and the general form of this deck when a library code file is to be produced, is in the disposition specified on the COMPILE card. The general format of the required COMPILE card to initiate a compilation for library is:

```
<I>COMPILE program name WITH FORTRAN LIBRARY
```

A disposition of LIBRARY indicates that the object code file resulting from a successful compilation be placed on the library with an assigned SAVEFACTOR (in this case, 30 days) and a TITLE identical to the program name or the program name modified by subroutine names in the case of separate compilation. This latter case involves the compilation of a group of program units with the SEPARATE compiler option set so that these program units may be bound as desired at a later time. (See the Program Binder document.) Each separately compiled program unit is assigned a separate code file; the TITLE of each of these separate code files consists of the program name with the rightmost identifier of the name replaced by the subprogram name (or unmodified in the case of a main program unit).

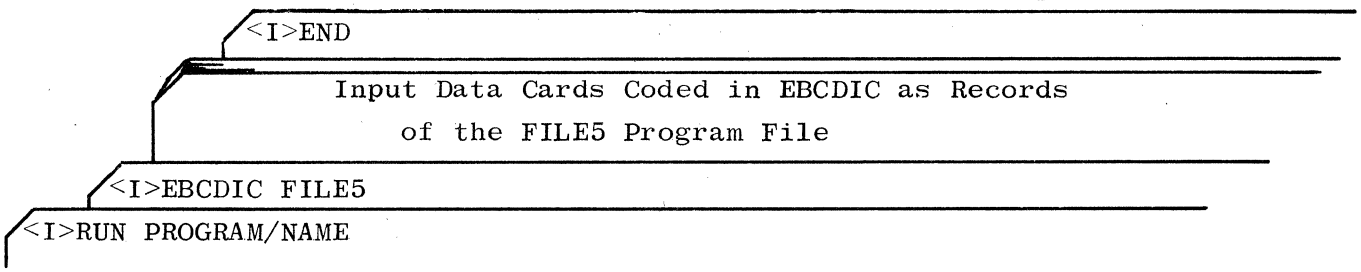
The library code file resulting from a standard compilation may be executed via the RUN or EXECUTE system control statement. An execution deck consists of (in order):

a. A RUN or EXECUTE system control card bearing the TITLE of the code file to be executed.
b. Optional MCP control cards which specify various execution parameters (e.g., FILE cards, PROCESSTIME cards, etc.)

c. An input data deck (if required for program execution) preceded by an EBCDIC or BCL control card (depending on the character code employed on the data cards) bearing the TITLE of the program input file involved. (Note: the default TITLE of a program file is the internal name of that file.)

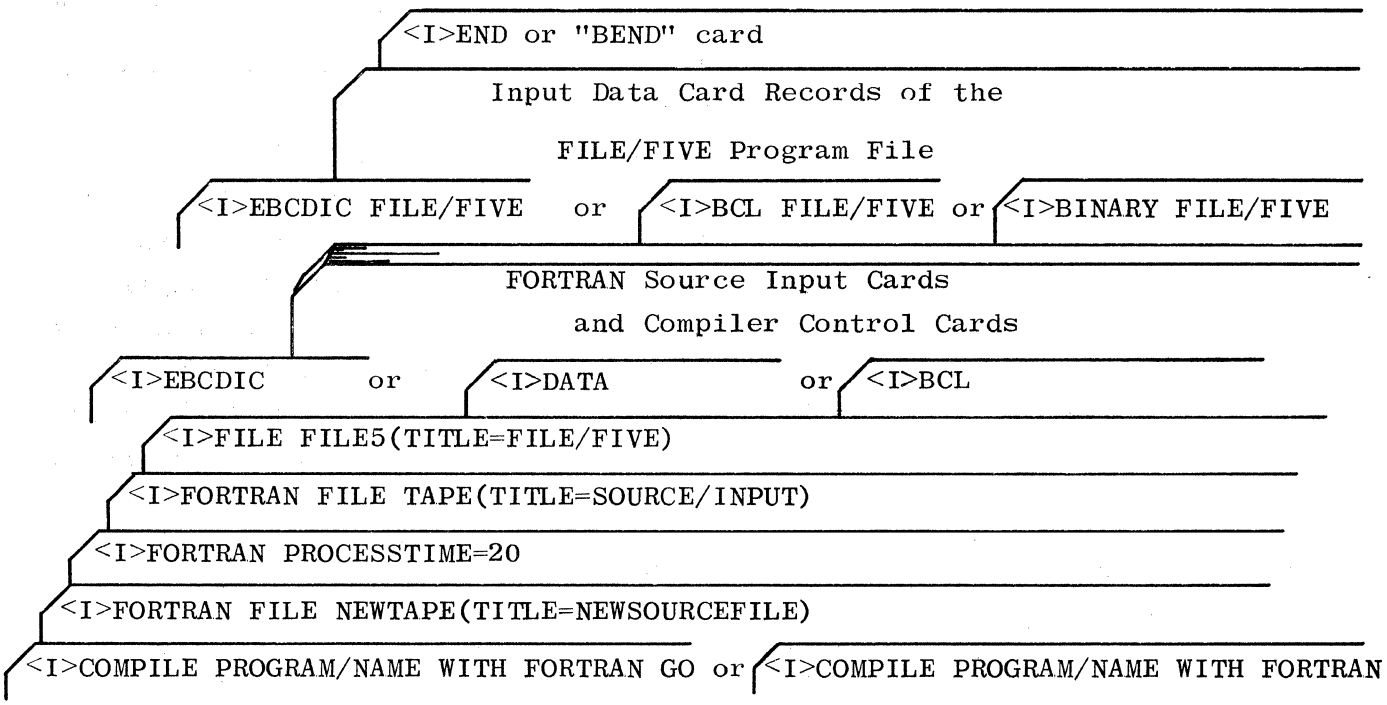
d. An END system control card to terminate the deck.

The basic execution deck is displayed in this example deck:



Compile and Go

The following sample deck displays the general deck format to be employed when a FORTRAN compilation is to be initiated to generate an object code file which is to be immediately executed and then discarded.


<I>END or "BEND" card
Input Data Card Records of the
FILE/FIVE Program File
<I>EBCDIC FILE/FIVE or <I>BCL FILE/FIVE or <I>BINARY FILE/FIVE
FORTRAN Source Input Cards
and Compiler Control Cards
<I>EBCDIC or <I>DATA or <I>BCL
<I>FILE FILE5(TITLE=FILE/FIVE)
<I>FORTRAN FILE TAPE(TITLE=SOURCE/INPUT)
<I>FORTRAN PROCESSTIME=20
<I>FORTRAN FILE NEWTAPE(TITLE=NEWSOURCEFILE)
<I>COMPILE PROGRAM/NAME WITH FORTRAN GO or <I>COMPILE PROGRAM/NAME WITH FORTRAN

The first card of the program deck is the required COMPILE system control card which bears the program name (in this case, PROGRAM/NAME) and a disposition of GO or a blank disposition specification, either of which indicates that a compilation and subsequent execution of the resultant code file (in the event of a successful compilation) are to be initiated.

The second card in this sample deck is a FILE system control card which specifies that the updated symbolic file produced by the compiler be assigned the TITLE NEWSOURCEFILE. The presence of this card is optional.

The next card is a PROCESSTIME system control card which specifies that a maximum of 20 seconds of processing time be allowed for this compilation. Since this deck does not contain another PROCESSTIME control card to specify a maximum processing time for the subsequent program execution (i.e., a PROCESSTIME card bearing no compiler name), the 20 second process time limit applies to the execution also. Such cards are optional.

The following sample card in this deck is a FILE system control card which indicates that the secondary compiler input file TAPE has the directory TITLE of SOURCE/INPUT. The presence of this card is optional.

The following card in this sample deck is an example of the use of a FILE system control card to specify the value of an attribute of a program file. (The fact that this card does not apply to a compiler file is denoted by the absence of the compiler name (i.e., FORTRAN) from this card.) The function of the card appearing here is to indicate that the program file with the internal name FILE5 is to use the file with the TITLE of FILE/FIVE. The TITLE is the only file attribute affected by this FILE card; such a system control card overrides file attribute values assigned within the FORTRAN program. Any TITLE assigned to this file within the program being compiled will be ignored, and FILE/FIVE will be used instead of any other TITLE.

The following card in the deck shown is either an EBCDIC or a BCL system control card, depending on the character code used on the following input cards. If BCD coding is used on the following input cards, then the BCL control card should be used and the BCD compiler option must be set prior to the appearance of the first BCD-coded input card. The present MCP control card is used to label the following input card deck. No file TITLE appears on the EBCDIC or BCL card as the following cards are automatically assigned as records of the compiler input file CARD.

The next item in this sample deck consists of the compiler input cards. These cards (up to the EBCDIC or BCL card which is displayed next) are the records of the CARD compiler input file and bear FORTRAN program statements and compiler control statements. Each of the cards in this file may have eight-digit sequence numbers in card columns 73 through 80. These sequence

numbers are used when CARD and TAPE file input records are to be merged. The compiler CARD file could have been label-equated to another medium (e.g., DISK), in which case the present input cards and the preceding EBCDIC or BCL card could have been omitted. Also, the TAPE file could provide the sole source input, with the CARD file containing only a record setting the MERGE option.

The following card in this sample deck is either an EBCDIC or a BCL system control card used to label the following input cards; both need appear only if the program which is to be executed requires input cards for one of its files. The present control card must bear the TITLE of the appropriate program file (e.g., FILE/FIVE). The program file with this TITLE must be an input card file (i.e., UNIT = READER). Since a FILE system control card appears earlier in this sample deck which equates the TITLE of FILE/FIVE to the internal file name FILE5, the input cards which follow this EBCDIC or BCL card are records of the FILE5 file.

The following item in the sample deck consists of the input data card records of the file with the TITLE of FILE/FIVE. These records are punched in the card code indicated on the preceding system control card (i.e., EBCDIC or BCL).

The final card in this deck is the END system control card which must terminate the deck.

A Compile-and-Go deck may contain as many EBCDIC- or BCL-labeled program execution input data decks as are required by the program input card files. These decks must follow the source input cards required for the program compilation, and the label card (i.e., the EBCDIC or BCL card) of each of these data decks must bear the TITLE of the program card input file to which the following card records are to be assigned. A binary card file must always be terminated by a binary end MCP control card.

Compile for Library and Go

A B 6700/B 7700 FORTRAN program may be compiled with the intention of immediately executing the generated object code file and then retaining that code file among the library files. The basic difference between the "Compile-and-Go" deck set-up and the general form for the deck to be used when the function just described is to be performed is in the disposition specified on the COMPILE system control card initiating the deck. The general format of the required COMPILE card for the "Compile-for-Library-and-Go" case is:

```
<I>COMPILE program name WITH FORTRAN LIBRARY GO
```

A COMPILE card disposal of LIBRARY GO indicates that the object code file resulting from a successful compilation be placed on the library with an assigned SAVEFACTOR (in this case, 30 days) and a TITLE identical to the program name and be also immediately placed into the job mix and executed after the compilation. An error condition will exist if a separate compilation is attempted using this COMPILE card disposition; a separately compiled program unit may not be executed without binding. An exception is a main program which calls no subprograms.

Decks of input data cards required by program files during execution may be contained in the present deck in the same manner in which they may be contained in a "Compile-and-Go" deck.

BATCH FACILITY

In certain situations, such as an educational environment, it may be possible to submit as a group a number of programs having the following characteristics:

a. The programs are to be compiled for syntax or compiled and executed.
b. If a program has no syntax errors and execution is requested, the execution time is relatively small.
c. Each program requires no more than one printer file and no more than one card reader file.
d. No program requires operator intervention.

The batch facility is available so that groups of these programs may share the cost of many required system overhead functions normally associated with each job, such as job initialization and termination, linking with intrinsics, memory allocation, etc. Sharing these tasks promotes better use of system resources.

Deck Set-Up

Each job, which must conform to the restrictions discussed below, is set up as follows. All $ cards must have the format of $ option cards as described in chapter 19, i.e., the $ must be in column 1 or the $ must be in column 2 with a blank in column 1.

```
$JOB  optional job title
    program
$ENTRY
    data, if any
```

The first card in the deck is a $JOB card, which is used to indicate the beginning of the program. Optionally, a job title may appear on the $JOB card; this

title is used to identify printed output produced by the compiler. Following the $JOB card is the source program, complete with the normal collection of compiler option cards, etc.

The next item is the $ENTRY card. This card is used to indicate the end of the program. It is also used to indicate that execution is desired if there were no errors. If the $ENTRY card is missing, the compiler assumes that no execution of this program was wanted, i.e., only a compile for syntax was requested.

Finally, if the program uses a data deck, then the data deck appears after the $ENTRY card. Note that the $ENTRY card is required for execution regardless of whether or not there is a data deck.

In addition, there are two compiler options which apply to the execution of each job. The two options, PROCESSTIME and IOTIME, perform the same functions as the corresponding system control cards.

For example:

```
$ PROCESSTIME = 2.2 PROCESSIOTIME = 5
```

would result in an upper limit of 2.2 seconds on execution time and 5 seconds in I/O time. There is no way to limit elapsed time, since programs cannot control the elapsed time.

If PROCESSTIME or IOTIME is set prior to the first $JOB card, the values become upper limits to the PROCESSTIME and IOTIME for individual user programs. Any individual user may be restricted to lower limits by the inclusion of PROCESSTIME or IOTIME compiler option cards following the $JOB card.

Using the Batch Facility

The job decks are placed together to form a single deck. This combined deck is preceded by the usual system control cards invoking the FORTRAN compiler and followed by the <I>END system control card. This single deck is then processed by the system. The output produced is a single printer file having each compiler listing followed by its corresponding printed output.

Example of a batch of three FORTRAN jobs:

```
<I>COMPILE BATCH FORTRAN;DATA
$PROCESSTIME = 5.5 IOTIME = 1
$JOB                JOB # 1
$SET FREE RESET SINGLE
REAL A(10)
READ 1,A
PRINT 1,A
1     FORMAT(10A6)
END
$ENTRY
THIS IS A DATA CARD FOR THE ABOVE JOB
```

```
$JOB                JOB # 2
         WRITE(6,1) (SIN(3.14*I),  I=1, 20)
1      FORMAT(1H , F13.4)
       I=X + + X       % SYNTAX ERROR
       END
$ENTRY

$JOB                JOB # 3
C***           COMPILE FOR SYNTAX
       SUBROUTINE XXXXXX
       DO 10 I=1,10
10      CONTINUE     % A SILLY LOOP
       RETURN
       END
       X=Y
$SET CODE
       CALL XXXXXX
       STOP
       END      % THERE IS NO $ENTRY CARD -- JUST A COMPILE FOR SYNTAX

<I>END
```

NOTE

No job in this three-job batch will be allowed more than 5.5 seconds of processor time, nor more than 1 second of I/O time. Note also that job #1 will compile correctly and run, using the one-card data deck immediately following its $ENTRY card; that job #2 has a syntax error, but would have run had it been error free (it has no data deck, however); and that job #3, despite being syntactically correct, will not run without an $ENTRY card.

Restrictions

Each of these restrictions must be met in order for a job to be a candidate for successful processing by the batch facilities of the FORTRAN compiler.

a. The PRINTER file may not be explicitly opened, closed, or have its attributes changed. Attempts to do this will terminate the job.
b. Binding is not allowed. No compiler option pertinent to binding is valid.
c. Missing subroutines or functions are fatal errors.
d. The PAUSE statement is allowed but has no effect.

e. Any program action requiring operator intervention is a fatal execution error.
f. Only one printer file is allowed. If two or more printer files are declared, their output will be joined into a single file. Similarly, at most, one reader file is allowed.
g. The following compiler option cards are invalid or ignored when using the batch facilities: BCD, END, ERRLIST, LIBRARY, LINEINFO, NEW, NEWSEGMENT, SEPARATE, TIME, XREF.
h. All job decks must be punched with the same card codes (EBCDIC, BCD, BCL, or ASCII). The system control card preceding the job decks (DATA, EBCDIC, ASCII, or BCL) must also indicate the proper card code.

COMPILATIONS THROUGH CANDE

A FORTRAN compilation initiated using the Command and Edit (CANDE) language differs from other FORTRAN compilations in the following ways:

a. Compiler option LIST is initially reset.
b. Compiler options FREE, ERRLIST, and LINEINFO are set.
c. Compiler option LIMIT has a default value of 10.
d. The ERRORFILE into which syntax errors are written is equated to the remote terminal initiating the compilation.
e. Unless otherwise specified:
    1. The CARD file will be equated to the work file or specified disk file.
    2. No TAPE file is used.
    3. The object file created by compiling the code file <filename> is USERCODE/<usercode>/OBJECT/<filename>.
f. All compilations are of the COMPILE FOR LIBRARY type. Execution of the object code may be specified by either of two CANDE commands: RUN or EXECUTE.

Language Compatibility

FORTRAN IV, Level H

Source language programs written in FORTRAN IV, Level H, are acceptable as input to this compiler, with the exceptions stated below. The following statements relating to these exceptions and other differences refer to the features of the B 6700/B 7700 FORTRAN compiler.

a. The number of continuation cards is not limited.

b. The characteristics of allowable constants are as follows:

(1) Single precision: an integer or the mantissa of a real may be from one to 11 decimal digits. The approximate total range of a real is 0 and 8.758E-47 to 4.314E68.

(2) Double Precision: an integer or the mantissa of a double may be from one to 23 decimal digits. The approximate total range of a double is 0 and 1.939D-29581 to 1.949D29603.

(3) Complex: The characteristics of each component of a complex number are the same as those of a single-precision number. A component of a complex number that is outside the range of a single-precision number will be rounded or disallowed, as necessary. Each double-precision complex intrinsic identifier is treated as a synonym for the corresponding single-precision complex intrinsic identifier.

c. Variables local to a SUBROUTINE or FUNCTION subprogram will not retain their values between calls on that subprogram unless the compiler option OWN is set in an appropriate manner for the subprogram. (See also point m.)

d. The indexing parameters of a "DO" statement may be changed with the range of the DO loop.

e. Transfer out of and into a DO loop is unrestricted.

f. A "STOP n" statement is allowed, but n will not be displayed.

g. The "DEFINE FILE" statement is not recognized. The functions of this statement are included in the file declaration statement or a FILE system control card.

h. An "end-of-file action" specifier (,END = <label>) is allowed in a RANDOM I/O (direct access) statement.

i. The "optional asterisk" specification in a type statement is allowed, but it will have no effect except that "REAL *8" will designate DOUBLE PRECISION.

j. Array elements may appear in the expression in an arithmetic statement function.

k. DATA and EQUIVALENCE statements may contain array names as elements of the list of data items affected by the statement. Such elements are not restricted to array elements and simple variables.

l. Recursion is allowed; that is, a subprogram may reference itself either directly, indirectly, or through an entry point.

m. An entry into a subprogram at any point results in initialization of only those formal parameters included in the formal parameter list of the FUNCTION, SUBROUTINE, or ENTRY statement through which the subprogram was entered. Of the remaining formal parameters, those which are simple variables are reset to zero, and those which are arrays are nonexistent. All local variables not appearing in a DATA statement or initialized in a type declaration are reset to zero.

n. Any FORTRAN-supplied in-line function specified in an EXTERNAL statement will not be expanded in line. Such a function is assumed to be user-supplied and will be treated as a function subprogram.

o. Data in unlabeled common may be initialized by a BLOCK DATA subprogram.

p. The COMMON statement in a BLOCK DATA subprogram need contain only those elements of the block up to and including the last element to be initialized.

q. Elements of common blocks may be initialized with any of the following: (1) initial value lists, (2) DATA statements in the main program, or (3) a BLOCK DATA subprogram.

r. Strings can be stored only as integer or real variables (six characters per identifier) or as double-precision variables (12 characters per identifier.)

B 3500 FORTRAN

Source language programs written in B 3500 FORTRAN are acceptable as input to this compiler, with the exceptions stated below. The following statements relating to these exceptions refer to this compiler.

a. Integer and real constants are limited to 11 significant digits. Double-precision constants are limited to 23 significant digits.
b. Adjacent exponentiations are normally carried out in right-to-left order.
c. A "STOP n" statement is allowed, but n will not be displayed.
d. Simple variables in a formal parameter list are "call by value" unless enclosed in slashes.
e. Execution of a "PAUSE n" statement results in suspension of execution of the program and display of n. Execution may be continued with an "OK" message from the operator's console.
f. The compiler options are different.
g. B 3500 extensions to ANSI FORTRAN such as the ALPHA type declaration are not accepted.

ANSI FORTRAN

Source language programs written in ANSI FORTRAN are acceptable as input to this FORTRAN compiler without exception.

Other FORTRAN Languages

The B 6700/B 7700 FORTRAN compiler was designed to take advantage of the similarities among FORTRAN languages. However, since some language features are machine-dependent and since variations and extensions in implementation exist, no guarantees of compatibility can be made.

The language portion of this document should be used as a tool in rendering source programs written in other FORTRAN languages acceptable as input to the FORTRAN compiler discussed here. Any language constructs which the user feels may possibly be incompatible with this FORTRAN compiler should be compared with valid B 6700/B 7700 FORTRAN constructs which perform an identical function. In general, incompatibility tends to arise in the following areas:

a. Differences among FORTRAN I/O statements and file description statements.
b. The inability to intersperse B 6700/B 7700 FORTRAN with statements in other languages.
c. Differences among FORTRAN string constructs reflecting internal data word structure.
d. Differences among FORTRAN intrinsic names.

e. Differences in the manner in which COMMON storage is handled.

f. The use of FORTRAN constructs which are extensions to both FORTRAN IV-H and ANSI FORTRAN.

The syntax error checking ability of the B 6700/B 7700 FORTRAN provides a valuable tool in the detection of such language incompatibilities.

CHAPTER 19

COMPILER OPTIONS

Compiler control statements may be inserted between FORTRAN statements to affect the manner in which the compiler processes source statements. This chapter discusses these options. Also described are some compiler generation options that control certain properties of the FORTRAN compiler when it is compiled from its ALGOL source. The FORTRAN programmer should ascertain the value of any relevant options for the compiler being used.

COMPILER CONTROL STATEMENTS

To a limited extent, the FORTRAN programmer may control the manner in which which the B 6700/B 7700 FORTRAN compiler processes the source input which it accepts. The user may specify the manner in which the compiler is to receive the source input, the consequences of certain syntax errors, and the form of the generated compiler output. The medium whereby these constraints are communicated to the compiler consists of the compiler control statement. Such statements may be input to the compiler upon a card in the same manner as source language statements. Compiler control statements are input to the FORTRAN compiler via compiler control cards which may occur at any point in the compiler input files and which may contain only compiler control information.

A compiler control card is identified by the appearance of a dollar sign ($) in the first or second column of the card. If the $ is placed in card column 2, and there is a blank in column 1, then the compiler control card image will be placed in the updated symbolic file (NEWTAPE) if such a file is generated. (An INCLUDE card will be placed in this file only if the option INCLNEW is reset.) The compiler control information is punched in the following columns through column 72, with an eight-digit sequence number in columns 73 through 80 (all blanks represents the lowest-valued sequence number). A compiler control card with no other compiler information causes the card image in the secondary input file with the same sequence number to be ignored.

The basic element of compiler control information is the compiler option, which may be invoked via the appearance of its name on a compiler control card. Two mutually exclusive states are associated with the majority of

these options: set and reset; various compiler functions are dependent upon the states of such options (called standard options.) Default states are assigned to these compiler options and the desired state of such an option may be specified on a compiler control card. Such control cards may also bear any arguments associated with some of the options. The balance of compiler options are parameter options with which no states are associated. The functions performed by these latter options are initiated by the appearance of the appropriate option name (and any related arguments) on a compiler control card.

Compiler control cards may contain only the following information items in addition to the initial $ and terminal sequence number:

a. Option actions which assign states to indicated standard options.
b. Option names and/or associated option arguments (i.e., literals, etc., which are connected with the function of the options).

These two types of compiler control information are discussed in the following paragraphs.

Compiler option cards differ from FORTRAN statements in that blanks are not ignored; they are used to separate each option or option action. Blanks may optionally appear between the $ and the first option or option action. Compiler option cards may not be continued.

Option Actions

The purpose of a compiler control statement may be to assign a desired state (i.e., set or reset) with an indicated standard compiler option(s). Such a control statement must begin with either an explicit or an implicit option action. An explicit option action is defined as one of these mnemonics:

SET
RESET
POP

An implicit option action is indicated when a compiler control statement contains only the names of standard options and no explicit option action. In the latter case, all standard options named in the control statement are assigned the state SET and all other standard options are assigned the state RESET.

If a compiler control statement begins with the option action SET, the options following the option action are assigned the state SET; the states of all other standard options will be unchanged. If the control statement begins

with the option action RESET, the options following the option action are assigned the state RESET; the states of all other standard options will be unchanged. If the specified option action is POP, then the options indicated will revert to their immediately previous states (their states will become RESET if these options have not been previously changed from their default states). The states of all other options will be unchanged. The following are examples of compiler control statements employing the SET, RESET, and POP option actions:

```
SET LIST SINGLE INCLNEW
RESET VOID
POP NEW NEWSEQERR
SET SEQ 0+100
```

A standard option which has a default state of RESET is initially assigned a 24-bit stack filled with zeros; an option which has a default state of SET is initially assigned a 24-bit stack with a 1 on top and zeros in the remaining positions. The top stack position denotes the state of the option at any time. Each SET option action causes the stacks allocated to the designated standard options to be pushed down one bit and a 1 to be placed on the top of each of these stacks. Each RESET causes the appropriate option stacks to be pushed down one bit and a 0 to be placed on the tops of these stacks. POP causes the stacks corresponding to the designated options to be popped up one bit, causing the associated options to revert to their immediately previous states. Since the size of these option stacks is 24 bits, a maximum history of 24 states may be recorded. When a compiler control card appears bearing a standard option name and an implicit option action, then the resultant action is identical to that which would have resulted had all 24 bits of each standard option stack been reset followed by an explicit SET performed on each indicated standard option. For example, after the appearance of a card containing this compiler control information:

```
$ CODE
```

the history stack for the CODE option contains a 1 in the top stack position and all zeros in the following positions. The history stack for each of the other standard compiler options (e.g., LIST, BCD, VOIDT, etc.) then contains all zeros. A compiler control statement which applies to standard compiler options begins with an explicit or implicit option action, contains a list of standard options to which the option action is to apply, and ends when the next explicit option action is encountered on the same compiler control card or when a percent sign or column 72 of the card is reached. The compiler options affected by

the control statement will retain the indicated states for all input cards with sequence numbers greater than the sequence number on the compiler control card bearing the control statement (or the physically succeeding input cards for a deck in which all sequence numbers are blank) until another compiler control card is encountered which alters the option states. A compiler control statement may also contain any parameter option name except INCLUDE and the action initiated by the appearance of the option name will still result. The following is a valid example of a card bearing compiler control statements employing option actions:

```
$SET SINGLE LIST SEQ 10+5 RESET OWN BCD                    00001070
```

The control statements just displayed assign the state SET to the options SINGLE, LIST, and SEQ (with the sequencing arguments of 10 and +5) and assign the state RESET to the options OWN and BCD. The card bears the eight-digit sequence number 00001070 in columns 73 through 80.

The appearance of a parameter option on a compiler control card with an implicit option action (i.e., no SET, RESET, or POP) does not result in the RESETting of any options. The names of these parameter options are themselves compiler control statements and may appear on a compiler control card with other compiler control statements, except for the INCLUDE option which must appear alone on a compiler control card.

NOTE

The appearance of a one- to eight-digit unsigned integer or such an integer preceded by a "+", except in columns 73-80, constitutes a parameter control statement used as an argument associated with the SEQ option.

It is possible to assign the state of one compiler option as the value of another compiler option; for example:

```
$ SET SINGLE = LIST
```

The B 6700/B 7700 FORTRAN compiler recognizes the following identifiers as valid compiler option names (a (p) denotes a parameter option):

ASCII	
AUTOBIND	
BCD	
BCL	
BINDER	
B5500	
B5700	
B7700	
CHARS	(p)
CHECK	
CODE	
DBLTOSNGL	
DEBUG	
END	
ENTRY	
ERRLIST	
FREE	
FREENEW	
FREETAPE	
GRAPH	
INCLNEW	
INCLUDE	(p)
INSTALLATION	
IOTIME	(p)
JOB	
LEVEL	(p)
LIBRARY	
LIMIT	(p)
LINEINFO	
LIST	
LONG	
MERGE	
MONITOR	
NEW	
NEWSEGMENT	(p)
NEWSEQERR	
NOBINDINFO	
NOWARN	
OMIT	

COMPILER OPTIONS

```
OMITDEBUG
OPT             (p)
OWN
OWNARRAYS
PAGE            (p)
PRINTER         (p)
PRINTLIMIT      (p)
PROCESSTIME     (p)
READER          (p)
SEGMENTATION    (p)
SEPARATE
SEQ
SEQERR
SINGLE
STACK
SUPRS
TIME
TRACE
VECTORMODE
VOID
VOIDT
XREF
$
```

The appearance on a compiler control card of any compiler option name not contained in the preceding list constitutes a compilation error.

The FORTRAN compiler options are discussed alphabetically in the following paragraphs. The default state of each option is indicated in parentheses following the option name; the function performed by the option is discussed in the paragraph accompanying the name. (The default state of the LIST option is SET and the default state of the balance of the standard options is RESET unless the compiler is employed via the CANDE language. If the compiler is called from CANDE, the default state of the ERRLIST, FREE, and LINEINFO options is SET and the default state of the balance of the standard options is RESET.) The FORTRAN compiler options are described in the following paragraphs.

- **ASCII (Reset)**

The ASCII compiler option allows the compiler to accept source programs in ASCII code.

- **AUTOBIND (Reset)**

The AUTOBIND compiler option combines the processes of compiling and program binding into one job. During compilation, the compiler produces a set of instructions to be passed to the binder. In many cases, these binder instructions are self-sufficient for binding purposes; however, the user may insert his own binder control cards where instructions are required. The AUTOBIND option should be zet or reset only once, at the beginning of a compilation. (Refer to Program Binder Manual, Form No. 5000045.)

During binding the main program will serve as the host for binding. At most, one host may be compiled in a job along with any number of separate subroutines. If no eligible host is being compiled, a host must be specified. Two methods are available:

<I> COMPILER FILE HOST (TITLE = <filename>)

or a binder host card, such as:

$ HOST IS <filename>;

however, the above two methods will not override the use of the main program as the host file for binding.

Two types of control information may be passed to the binder when using the autobinding feature. Binder option cards (i.e., dollar cards) are input by a "$ BINDER" followed by the list of options to be reset or set. Other than the word "BINDER" following the dollar sign, the card is written exactly as if it were a dollar card input directly to the binder.

Binder input statements are passed to the binder by placing a dollar sign before them. Other than the dollar sign, the input statement is written exactly as if it were input directly to the binder, including a semicolon at the termination of the statement.

Example:

The following input deck is used to compile a main program named MAIN and a subroutine named SUBR, and replace an existing routine named OLDSUBR with the newly compiled SUBR. The program is then executed. The parameters of SUBR and OLDSUBR must match in number and type. If the subroutines had the same name, the "$ USE" card would not be necessary:

COMPILER OPTIONS

```
<I> COMPILE MAIN/PROG FORTRAN LIBRARY GO
<I> DATA
 $  SET SEPARATE
 $  SET AUTOBIND
 $  USE OLDSUBR FOR SUBR;
     MAIN PROGRAM
          .
          .
          .
     SUBROUTINE SUBR
          .
          .
          .
<I> END
```

- **BCD (Reset)**

The BCD compiler option allows the user to expand the compiler input file handling capability. The FORTRAN compiler accepts 6-bit BCD-coded (i.e., Hollerith-coded) characters as input if the compiler input file is labeled using a BCL system control card and the BCD option is assigned the state SET for the region of the file in which the record(s) bearing BCD-coded data appears. The BCD option may be set and reset as necessary within the input file. Compiler control cards appearing in the file at a point for which the BCD option is set must also be BCD-coded.

- **BCL (Reset)**

The BCL compiler option allows the compiler to accept source programs in BCL code. Setting compiler options B5500 or B5700 causes this option to be set.

- **BINDER (No associated states)**

See the AUTOBIND option.

- **B5500 (Reset)**

The B5500 and B5700 compiler options allow B 5500/B 5700 source programs to be compiled, with the exceptions listed below. These options cause the options BCL and CHARS = 6 to be set.

a. Some compiler options are different.
b. With respect to "call by name" and "call by value", all formal parameters are handled in the manner described in chapter 15 of this manual.

c. Actual parameters used for nonstandard returns must have the form "&n" instead of "$n".
d. No alternate for ".EQ.", ".NE.", ".GE.", or ".LE." is recognized.
e. Adjacent exponentiations are normally evaluted in right-to-left order.
f. Common blocks may not be extended backward by EQUIVALENCE statements.
g. If no file declaration card appears with "READER" instead of a file number, the forms "READ (5, n)" and "READ n," are equivalent.
h. If no file declaration card appears with "PRINTER" instead of a file number, the forms "WRITE (6, n)" and "PRINT n," are equivalent.
i. If no file declaration card appears with "PUNCH" instead of a file number, the forms "WRITE (7, n)" and "PUNCH n", are equivalent.
j. There is no "LOCK" option on the file declaration card.
k. The first character of each NAMELIST input record is ignored. The first record must begin with one blank followed by & and then followed by the NAMELIST name, and finally followed by at least one blank. The last record must be terminated with &END.
l. Executable and nonexecutable statements may not be mixed on the same source card

- **B5700 (Reset)**

This option is the same as the B5500 option.

- **B7700 (Reset)**

When this option is set and OPT=1, the compiler will generate code better suited for the B 7700. (Refer to chapter 20, OPTIMIZATION AND VECTORMODE.)

- **CHARS (No associated states)**

The compiler option CHARS=n causes all strings to be stored in the right-most n*8 (or n*6 for BCL and BCD) bits of each word. The default value of CHARS is 6 for EBCDIC characters. Setting compiler options B5500 or B5700 causes CHARS to be set to 6.

- **CHECK (Reset)**

The CHECK compiler option allows the user to expand the compiler error-checking ability. When set, the CHECK option causes the compiler source input files and updated source file (NEWTAPE) to be checked for sequence number errors. A sequence error condition exists when a sequence number on a source card or card image is less than the sequence number on the preceding card. A warning message is placed on the output listing when such an error is found. When the CHECK, SEQERR, and NEWSEQERR options are reset, the compiler performs no sequence checking.

COMPILER OPTIONS

- **CODE (Reset)**

The CODE compiler option allows the user to specify that compilation information beyond the basic items be included in the output printer listing. When set, the CODE option causes the output listing to contain the object code generated by the compiler. LIST must be set to produce this listing.

- **DBLTOSNGL (Reset)**

When set, this compiler option causes all double precision quantities to be treated as single precision.

- **DEBUG (Reset)**

The DEBUG compiler option allows the user to specify that the printer listing is to contain extensive compilation information. When set, the DEBUG option causes lists of stack allocations, names of scanned items, and indications of compiler procedure entries and exits to be included in the output listing. The function of this option may change without notice.

- **END (No associated states)**

The END compiler option signals an end of file on the medium where it is encountered.

- **ENTRY**

This is used only with the batch facility and is described in chapter 18.

- **ERRLIST (Reset for non-CANDE compilations, set for CANDE compilations)**

The ERRLIST compiler option is provided primarily for use with compilations initiated through CANDE to allow the listing of compilation errors separate from the listing of the source input. When set, the ERRLIST option causes an error listing to be generated on the compiler output file ERRORFILE. When a compilation error is detected in the source input, the offending line of text, an error message, and a pointer to the syntactical item in question are written on two lines in the ERRORFILE file. This option is provided primarily for use when the FORTRAN compiler is called from a remote terminal by the CANDE language, but it may be used regardless of the manner in which the compiler is called. When the compiler is called from CANDE, the default state of the ERRLIST option is SET and ERRORFILE is automatically equated to the remote device involved.

- **FREE (Reset for non-CANDE compilations, set for CANDE compilations)**

The FREE compiler option allows the compilation of free-form source programs from the source file CARD. One purpose of this option is to allow simple input from remote devices. The following restrictions must be observed:

a. Continuation statements must contain a dash (-) or asterisk (*) as the first non-blank character in the first six columns.
b. Comment statements must have a "C" in column one, and a dash or asterisk in column 2.
c. For statements beginning with labels, the label number must appear in the first five columns.
d. Each free-form statement may contain 66 characters, at most, including blanks and not including sequence numbers. In the case of labelled statements or continuation statements, this restriction applies to those characters following the label or continuation dash or asterisk. Any characters beyond the 66th character but before column 73 will be ignored.
e. Columns 73 thru 80 of the source card image are reserved for the sequence numbers, as in the case of formatted source input.
f. Compiler option statements must have a dollar sign ($) in column one or a blank in column 1 with a dollar sign in column 2.
g. FILE and DEBUG statements must have the strings FILEbb and DEBUGb, respectively in columns 1-6, where b represents a blank.

- **FREENEW (Reset)**

When FREENEW and NEW are set, the updated symbolic output file NEWTAPE will be created in a form that can be read by a compiler with FREE set.

- **FREETAPE (Reset)**

This option permits the source file TAPE to be in free format (see the MERGE option).

- **GRAPH (Reset)**

When this option is set and OPT=1, a flow chart of the source program will be created (see chapter 20).

- **INCLNEW (Reset)**

The INCLNEW compiler option is provided as an aid in controlling the contents of the updated symbolic compiler output file. When set, the INCLNEW option causes source text which is input to the compiler via the INCLUDE compiler option to be output to file NEWTAPE if the option NEW is also set. If INCLNEW

is reset, then the INCLUDEd text will not be placed on the NEWTAPE file; however, $INCLUDE cards contained in the CARD and TAPE files will be output to NEWTAPE when the $ is in column 2. If the state of the NEW option is RESET, the state of INCLNEW is ignored.

- **INCLUDE (No associated states)**

The special action compiler option INCLUDE allows the user to input text to the FORTRAN compiler from specified files in addition to the CARD and TAPE files. When the compiler is accepting source input from the CARD and/or TAPE file and a $INCLUDE card is encountered, the indicated portion of the file denoted on this control card is inserted into the input text in place of the card bearing the INCLUDE control statement. The included text may also contain INCLUDE control cards, but only five levels of such text nesting are permissible.

The following are proper forms of the INCLUDE compiler control statement:

```
INCLUDE
INCLUDE m
INCLUDE m - n
INCLUDE - n
INCLUDE f
INCLUDE f m
INCLUDE f m - n
INCLUDE f - n
```

In the INCLUDE statement examples just presented, m is either an unsigned integer constant of from 1 to 8 digits or an *. This item indicates the sequence number of the first card image to be included from the file referenced by the present INCLUDE statement. If m is an * rather than a sequence number, then inclusion will begin with the first card of the file or the card following the last card included from this file by a previous INCLUDE card at the same level as the present INCLUDE card. If this item is missing, inclusion will begin at the same point indicated by the use of an *. The item n denotes a one-to-eight digit unsigned integer constant which is the sequence number of the last card image to be included from the file referenced by the present INCLUDE control card. If this item is missing, card images are included from the file until the end of the file is encountered.

The item f denotes either an internal file name (1 to 17 alphanumeric characters long) or a file TITLE enclosed in apostrophes (') which indicates the file from which the included card images are to be drawn. If this item

is an internal file name, a FILE system control card may be used to specify the TITLE and other attributes of the file. (The default TITLE of this file is the internal name employed and the default KIND is DISK; see chapter 18, USER/COMPILER INTERFACE.) If this item is missing from the INCLUDE statement, then the file to be employed is the file designated by the last INCLUDE card at that level. An error condition exists if an INCLUDE card appears which does not specify a file and no INCLUDE card specifying a file has preceded this card at the same level. Thus, the first INCLUDE compiler control card at any of the five possible levels of text nesting must contain either a file TITLE or an internal file name.

A compiler control card bearing an INCLUDE compiler control statement may contain no other control information besides this statement. The following are valid examples of INCLUDE compiler control cards:

```
$INCLUDE FILE8 00001000 - 09000000                    00569010
$INCLUDE                                              00570000
$INCLUDE 'SOURCE/XYZ' - 900                           00000100
$INCLUDE * - 00001000                                 00000180
$INCLUDE INCLFILE
```

The first INCLUDE card example presented here instructs the compiler to accept as input at this point of the input file all records from the library file indicated by the internal file name FILE8 beginning with the card image with the sequence number 00001000 (or the next higher sequence number) and terminating with the card image with the sequence number 09000000 (or the preceding lower sequence number if no card image in the file bears this number). The TITLE of the FILE8 file may be specified on a FILE system control card following the COMPILE control card initiating the compilation; the general form of such a FILE card is displayed in this example:

```
<I>FILE FILE8 (TITLE=SOURCE/INPUT/INCL)
```

The card just displayed indicates that the TITLE of the included file is SOURCE/INPUT/INCL; the file is a disk file since the FILE card in this example does not change the value of the KIND attribute from the default specification.

The second of the sample INCLUDE cards presented instructs the compiler to accept as input at this point of the input file a portion of the file accessed by the last INCLUDE card in this deck at this level. The card images to be included consist of all records remaining in the include file follow-

ing the last record of that file accessed by the preceding INCLUDE card referencing that file. If, for example, this sample card is in the same deck as the first sample card presented and no other INCLUDE cards intervene, the include file to be accessed is the FILE8 file and the first record to be included is the card image with the next sequence number after 09000000.

The third sample INCLUDE card presented instructs the compiler to accept as input at this point of the input file a portion of the file with the TITLE of SOURCE/XYZ. The card images to be included consist of all records remaining in the file between the last record included from that file at this level and the record with the sequence number immediately higher than 00000900.

The fourth sample INCLUDE card presented instructs the compiler to accept as input at this point of the input file a portion of the file accessed by the last INCLUDE card in this deck at this level. The card images to be included consist of all records remaining in the file between the last record included from that file at this level and the record with the sequence number immediately higher than 00001000. If, for example, this sample card is in the same deck as the previous sample card presented and no other INCLUDE cards intervene at this level, the records to be included consist of the remaining card images in the SOURCE/XYZ file up to the card with sequence number immediately higher than 00001000.

The final sample input card presented instructs the compiler to accept as input at this point of the input file a portion of the file with the internal name INCLFILE. The card images to be included consist of all records remaining in the file between the last record included from that file at this level and the end of the file.

Source files suitable for use by INCLUDE compiler control statements may be produced by the FORTRAN compiler via the NEWTAPE output file.

- **INSTALLATION (Reset)**

The INSTALLATION compiler option is provided to allow an application program to employ installation-produced (nonstandard) intrinsics. When the INSTALLATION option is set, the user may reference instrinsics which the FORTRAN compiler would not otherwise accept. This option may be set and reset as desired only upon compiler control cards appearing before the first card image bearing source language input, but the state the option is in at the point when the first source statement appears is the state which the option will retain for

the balance of the compilation deck. When the option is allowed to remain in its default state of RESET, only standard intrinsics may be employed by the program being compiled.

- **IOTIME (No associated states)**

This compiler option specifies a maximum I/O time for the execution of programs submitted under the batch facility, which is discussed in chapter 18.

- **JOB (No associated states)**

This compiler option is used with the batch facility discussed in chapter 18.

- **LEVEL (No associated states)**

The parameter compiler option LEVEL is provided for use with separate compilations to alter the lexicographical level at which a program unit is to be compiled. (See Program Binder Manual, Form No. 5000045.) The proper format for the LEVEL control statement is:

LEVEL n or LEVEL=n

where n is an unsigned integer constant greater than or equal to 3 and less than 31 which is the option argument indicating the desired lexicographical level. The LEVEL compiler control statement (when used) must appear immediately before the start of a program unit. Such a LEVEL statement applies to all program units between the card bearing this statement and the card bearing the next LEVEL control statement.

When the LEVEL statement appears before the first source language statement in the compilation deck, then the compiler option SEPARATE will automatically be assigned the state SET by the compiler for that deck. Before LEVEL control statements may appear preceding program units which do not begin at the start of the deck, the SEPARATE option must be set by a compiler control statement appearing before the first source statement in the deck.

The LEVEL option is used to override the lexicographical levels assigned to separately-compiled program units by the compiler. The default level assigned to main programs is 2, and the default level assigned to subprograms is 3. The LEVEL statement may not be used to assign a level of 2 to a program unit. When binding separately-compiled subprograms to separately-compiled program units written in such a language as ALGOL it is often desirable that the subprogram be compiled at a level higher than 3 in order to fit into the structure of the ALGOL program. For example, a separately-compiled FORTRAN subroutine may be bound in place of an ALGOL procedure which is declared within another ALGOL procedure in which case the bound-in program unit must

be compiled at a higher lexicographical level than 3. The LEVEL option is employed during the compilation of the subprogram to specify the necessary level. The internal function of the LEVEL option is to assign stack locations at the Dn level for all non-D2 stack or D1 stack program items.

- **LIBRARY (Reset)**

The compiler option LIBRARY is similar to the SEPARATE option except that all of the subroutines are compiled into one code file. The title of the resulting code file will be the title given the code file on the compile card. To modify one of the subroutines, all of them must be recompiled.

- **LIMIT (No associated states)**

The parameter compiler option LIMIT is provided to allow the user to indicate that the compilation is to be terminated when a certain maximum number of errors are detected in the source input. The proper format of the LIMIT compiler control statement is:

```
LIMIT n   or   LIMIT=n
```

where n is an unsigned integer constant which is the option argument specifying the maximum number of syntax errors which may be discovered in the source input before the compilation is prematurely terminated. A limit of 0 indicates that an "infinite" number of errors are to be allowed. The limit control statement may appear only on a compiler control card which precedes the first source language input statement. If no LIMIT statement appears, a default error limit of 200 is assigned unless the compilation is initiated through CANDE, in which case the default error limit is 10.

- **LINEINFO (Reset for non-CANDE compilations, set for CANDE compilations)**

The LINEINFO compiler option causes sequence number information and its relation to code emitted at compile time to be saved. If a program should then terminate abnormally, the sequence number of the point in the program where the error or termination occurred is provided.

- **LIST (Set for non-CANDE compilations, reset for CANDE compilations)**

The LIST compiler option is provided to allow the user to control the generation of the LINE output compiler file. An output listing is generated on the file LINE only if the LIST option is set for some portion of the source input or if the option TIME is set. This file is described in the discussion of "Compiler Files." The source statements appearing in the portions of the input files for which the LIST option is set will be placed on this printer listing. The LIST option may be set and reset as desired

via the appearance within the source input of compiler control statements altering the state of the option. When the LIST option is never set for a compilation (e.g., for non-CANDE compilations, when the LIST option is reset by a compiler control card which is the first card of the input deck) an output listing may be generated by setting the TIME compiler option; this listing will contain only compilation trailer information. Setting the TRACE compiler option automatically sets the LIST option.

- **LONG (Reset)**

The LONG compiler option is provided to allow the user to alter the manner in which the FORTRAN compiler handles array storage. Normally, program arrays requiring more than 1024 words of storage are segmented into 256-word segments for overlay purposes. Program arrays which are declared in portions of the source input for which the state of the LONG option is SET will not be segmented regardless of their storage requirements. (No program array may exceed 65,535 words in length.)

- **MERGE (Reset)**

The MERGE compiler option is provided to allow the user to specify constraints on the compiler ability to accept source input from more than one file. When the MERGE option is set, the compiler accepts source input from the input file TAPE in addition to source input from the file CARD. The primary input (from file CARD) is merged with the secondary input (from file TAPE) on the basis of sequence numbers to produce the total source input to the compiler. If matching sequence numbers occur on card images from these two files, the primary input card image overrides the secondary input card image. When MERGE is reset, only primary input is accepted by the compiler (i.e., from file CARD) and secondary input is totally ignored.

The total input to the compiler when MERGE is set consists of all card images from the file CARD, all card images from the file TAPE which do not have sequence numbers which may be found on cards in the CARD file, and all card images inserted into the text in these files by INCLUDE control cards. (Card images in the CARD file also override INCLUDE control cards in the TAPE file if sequence number conflicts are encountered.)

- **MONITOR (Reset)**

This compiler option allows the use of the MCP monitoring facilities.

COMPILER OPTIONS

- **NEW (Reset)**

The NEW compiler option is provided to allow the user to control a portion of the compiler's output file handling capability. When the state of the NEW option is SET, the merged input from the CARD and TAPE files will be placed on the updated symbolic output file NEWTAPE. Text inserted into the CARD and/or TAPE files will also be placed in the NEWTAPE file only if the INCLNEW compiler option is also set. Otherwise, if INCLNEW is reset and NEW is set, the INCLUDE cards will be placed on the NEWTAPE file rather than the INCLUDEd text. All compiler control cards beside INCLUDE in the merged CARD and TAPE input will be placed on the NEWTAPE file (when NEW is set) only if the initial $ on these cards is in card column 2.

The NEW option may be set and reset as necessary by compiler control cards appearing at any point in the input files. Such control cards may be also placed on the NEWTAPE file if the $ on these cards are in column 2.

The contents of the NEWTAPE file may be monitored in the following manner. When both the option NEW and the option LIST are set, the NEWTAPE file will contain all source language statements the LINE file contains (depending upon the state of the INCLNEW option) and all compiler control cards appearing in LINE which have their initial $ in card column 2. INCLUDE compiler control cards rather than included file text will be placed in NEWTAPE when the INCLNEW option is reset, but the included text will be placed in the LINE file regardless of the state of INCLNEW.

The NEWTAPE file will be created despite the occurrence of syntax errors in the source input; this file may be used as a secondary input for a later compilation or as an INCLUDEd file.

If the NEW compiler option and the FREE or FREETAPE compiler options are set concurrently, the compiler will perform a minor amount of editing on the output to the NEWTAPE file to ensure that the updated symbolic is acceptable as either free or fixed form input.

A title, other than the default title of FORSYM, may be specified for the NEWTAPE file by following the NEW option with a title enclosed in parentheses.

- **NEWSEGMENT (No associated states)**

The parameter compiler option NEWSEGMENT allows the user to alter the manner in which the FORTRAN compiler performs code file segmentation. Automatic segmentation of the output code file occurs when the size of a program unit exceeds a specified limit. (See section on Output Files, chapter 18.) The appearance of the NEWSEGMENT compiler control statement on a card image within

the source deck causes the code generated for the following source statement(s) to be placed in a new code file segment. This additional segment must be fetched from overlay memory (if not resident) and branching is thus necessitated at this point of the program execution. Therefore, the use of this control statement should be avoided at points in the code frequently passed during execution such as within loops. The proper format of the NEWSEGMENT control statement is:

NEWSEGMENT

No argument is associated with this option. This option allows segmentation to occur at a controlled point in the code file. When separate compilation is involved, this option may be employed only within main programs.

- **NEWSEQERR (Reset)**

The NEWSEQERR compiler option is provided to allow sequence number error checking to be performed on the updated symbolic output file as the file is generated. When the NEW and NEWSEQERR options are set, the updated symbolic file NEWTAPE is generated and the sequence numbers on the card images in this file are checked until a sequence number is found which is equal or less than the sequence number on the preceding card image. When such a sequence error is detected, a warning error message is placed on the printer listing and the NEWTAPE file is not locked after it is produced and will not remain on disk.

- **NOBINDINFO (Reset)**

When set, the compiler will not generate binding information with the code file; hence, the resulting code file cannot be processed by the binder.

- **NOWARN (Reset)**

The NOWARN compiler option allows the user to alter the compiler's error-handling facility. Warning syntax error messages may be produced when the state of the NOWARN option is RESET. When the NOWARN option is set, errors which otherwise would have generated warnings are considered syntax errors. For example, when NOWARN is reset, the first appearance of a variable name containing more than six characters generates a warning message on the printer listing. When NOWARN is set, the appearance of such a variable name is considered to be a syntax error.

- **OMIT (Reset)**

When set, this option causes card images from both CARD and TAPE files to be ignored by the compiler. They are, however, listed and/or included in a new

source file. On a listing, the omitted statements are flagged by the word OMIT. See also the VOID and VOIDT options.

- **OMITDEBUG (Reset)**

When this option is set, all debugging statements and their continuations are omitted.

- **OPT (No associated states)**

The parameter compiler option OPT allows the user to alter the form of the object code generated by the compiler by altering the manner in which the compiler handles various source input statements. The proper format of the OPT compiler control statement is:

```
OPT n   or   OPT=n
```

where the item n is the option argument, which is one of the integer constants 0, 1, or -1. The default value assigned to OPT for a compilation is 0.

A detailed description of optimization appears in chapter 20.

- **OWN (Reset)**

Subroutine variables which are not formal parameters and which are declared or first appear in a portion of source input where the FORTRAN compiler option OWN is set are allocated locations in the D2 program stack. The result of this action is to allow a subprogram variable to retain the value it held at the end of the last call on the subprogram when that subprogram is called again.

For example, consider this subroutine portion of a source language input file:

```
      SUBROUTINE AUGMNT (/X/,/Y/,Z)
      DIMENSION Z(5), HOLD(5)
$SET OWN
      INTEGER   A
      DIMENSION B(5)
      I=I+1
$RESET OWN
      X=X+I
      HOLD(1)=Y
      Y=Y+A
      A=HOLD(1)
      DO 1 J=1,5
      HOLD(J)=Z(J)
```

```
      Z(J)=Z(J)+B(J)
1     B(J)=HOLD(J)
      RETURN
      END
```

In this example, the OWN option is used to indicate that the variables A and I and the elements of the array B are to retain their values between subroutine calls. When the subroutine is first called, these items are initialized to zero as is normally the case with local variables. Thus, the function of this subroutine is to increment the value of X by 1 during the first call, by 2 during the second call, by 3 during the third call, etc. During each call, the variable Y and the elements of the Z array are increased by the values passed to these formal parameters when the AUGMNT subroutine was last called. During the first call on AUGMNT, these six data items are not increased.

A further restriction is that an array will be affected by the OWN option only if the specification statement (e.g., the DIMENSION statement) associated with the array lies in a region of code where the OWN option is set.

- **OWNARRAYS (See text)**

When the OWNARRAYS option is set, all arrays (and only arrays, unless the OWN option is also set) are treated as if they had been declared with OWN set. If array reinitialization is not required, faster execution will be realized if a program is compiled with OWNARRAYS set.

The default value of OWNARRAYS is the same as the compiler generation option SETOWNARRAYS discussed later in this chapter.

- **PAGE (No associated states)**

The parameter compiler option PAGE is provided to allow the user to partially control the format of the compilation output line printer listing. The appearance of the PAGE control statement on a compiler control card within the source deck causes the listing to be spaced to the top of the next physical page. The output listing then continues, beginning with the PAGE control statement. No argument is associated with the PAGE option. The following is an example of a compiler control card bearing the PAGE control statement:

```
$ PAGE SET SINGLE                                    90000010
```

- **PRINTER (No associated states)**

Option PRINTER=n causes each PRINT statement to employ file n.

COMPILER OPTIONS

- **PRINTLIMIT (No associated states)**

This option specifies a maximum number of print lines that may be generated by a program submitted under the batch facility described in chapter 18.

- **PROCESSTIME (No associated states)**

This option specifies a maximum processing time for programs submitted under the batch facility described in chapter 18.

- **READER (No associated states)**

Option READER=n causes each READ statement to employ file n.

- **SEGMENTATION**

The option SEGMENTATION=n modifies the algorithm described in chapter 18 (output files) for segmenting a program. Segment size n words will be used. In place of a number n, the characters MAX may be used. This will be interpreted as 65,535 words. The value of n may not exceed this limit.

- **SEPARATE (Reset)**

The SEPARATE compiler option is provided to allow the compilation of FORTRAN program units to produce separate code files which may be bound to other program unit files via the program binder. (See the PROGRAM BINDER Information Manual.) When the state of the SEPARATE option is SET for a "Compile for Library" compilation (see chapter 18), a separate object code file is generated for each program unit contained in the source input. The program units which may be handled in this manner are main programs, function subprograms, and subroutine subprograms. The code file generated for each of these program units will have a TITLE constructed from the program name contained on the COMPILE system control card initiating the compilation. The TITLE of the code file bearing the main program will be identical to the program name. The TITLE of the code file bearing a subprogram will be identical to the program name with the lowest-level (rightmost) identifier in the program name replaced by the name of the subprogram. For example, if the function with the name F is separately-compiled and the program name on the COMPILE card initiating the compilation is ABC/D, then the code file which is generated for F (if the compilation is successful) has the TITLE ABC/F. If the program name on the COMPILE system control card consists of only one identifier, then the TITLE of the code file bearing the subprogram will simply be the name of that subprogram.

SEPARATE may be set and reset as desired only upon compiler control cards appearing before the first card image bearing source language input, but the state the option is in at the point when the first source statement appears is the state which the option will retain for the balance of the compilation deck. Hence, one may only specify that all program units in the source input are to be separately-compiled or that all these program units are not to be separately compiled. The appearance of a LEVEL compiler control statement at the beginning of a source deck automatically sets SEPARATE for that compilation. LEVEL control statements may be embedded in a source deck only if SEPARATE is set for that deck.

- **SEQ (Reset)**

The SEQ compiler option is provided to allow the user to alter the form of the card images which are output to the line printer listing and to the updated symbolic file NEWTAPE. When the state of the SEQ option is SET, new sequence numbers are assigned to the card images placed in the listing and the NEWTAPE file (if any). This option has effect only when LIST and/or NEW are also set. The sequence numbers which appear on the card images in these files when the SEQ option is reset are identical to the sequence numbers on the corresponding cards in the input files.

When the SEQ option is set, sequencing begins with the default sequence number 00001000 and continues in default increments of 1000. These default sequencing parameters may be overridden in the following manner. The appearance of a one- to eight-digit unsigned integer on a compiler control card is assumed to be a special action control statement associated with the SEQ option when this integer is not immediately preceded on the card by the option name INCLUDE, LEVEL, LIMIT, or OPT. This integer will be employed as a sequencing argument when the state of the SEQ option is SET. If the integer is preceded by a plus sign (+), the integer will be used as the sequence number increment size. Otherwise, the integer will be used as the sequence number at which sequencing is to start as soon as SEQ is set. Both of these arguments may be specified, overriding the default values of 1000.

The sequencing arguments may appear on the same compiler control card as that setting the SEQ option, on a preceding control card, or on a later control card. The following are examples of a sequencing argument appearing on the same control card as the SEQ option:

```
$SET SEQ 100
$SEQ 20+1                                                    00800007
$RESET CODE SET SEQ LIST +200
```

In the first example, sequencing of the LINE and NEWTAPE files will begin at the sequence number 00000100 and continue in default increments of 1000 if no other sequencing increment was specified on a previous compiler control card. In the second example, sequencing will begin at the sequence number 00000020 and proceed in increments of 1. In the third example, sequencing will begin at the sequence number 00001000 (if this is the first time SEQ is set and no other initial sequence number was specified on a previous compiler control card) and proceed in increments of 200. Otherwise, sequencing will begin at a sequence number 200 greater than the last sequence number assigned, or at the initial sequence number assigned by an appropriate preceding control card.

The following is an example of a compiler control card specifying sequencing arguments but not setting the SEQ option:

```
$ 100 + 100                                                  00005000
```

This compiler control card only specifies that, when the state of SEQ is SET, sequencing is to begin with the sequence number 00000100 and proceed in increments of 100. (The state of no standard option is affected as this card bears parameter control information only and no standard option names.) If the SEQ option is set when this control card appears, these two sequencing arguments take immediate effect.

- **SEQERR (Reset)**

The SEQERR compiler option is provided to allow sequence number error checking to be performed on the source compiler input files. When SEQERR is set, the sequence numbers on the incoming source card images will be checked until a sequence number is found which is less than the sequence number on the preceding card. When such a sequence error is found, an error message is generated on the printer listing and the code file generated by the compilation will not be locked (i.e., will be discarded) regardless of the success of the compilation.

- **SINGLE (See text for default value)**

The SINGLE compiler option is provided to allow the user to alter the format of the line printer listing for the compilation. When the state of SINGLE is SET, the listing will be single-spaced; when the state of SINGLE is RESET,

the listing will be double-spaced. The default value of SINGLE is the same as the compiler generation option SINGLEBYDEFAULT discussed later in this chapter.

- **STACK (Reset)**

The STACK compiler option is provided to allow the user to increase the amount of compilation information contained on the printer listing. When the state of STACK is SET, the listing will contain the relative program stack addresses (address couples) of items named in the program being compiled. LIST must be set to produce the listing. The STACK option is automatically set when the TRACE compiler option is set.

- **SUPRS (Reset)**

The SUPRS compiler option is provided to allow the user to partially control the information contained in the printer listing for the compilation. When the state of SUPRS is SET, the appearance of warning messages in the output listing will be suppressed.

- **TIME (Reset)**

The TIME compiler option is provided to allow the user to specify that a minimal line printer listing is to be generated for the compilation. When the state of the TIME option is SET and the state of LIST is RESET, then the printer listing will be generated and will contain only compilation "trailer" information (e.g., number of cards scanned, number of errors, compilation times, etc.). No source cards will be listed (assuming LIST was reset for the entire input deck and no errors occurred). Since this trailer information is printed when LIST is set, the state of the TIME option is significant only when LIST is reset.

- **TRACE (Reset)**

The TRACE compiler option is provided to allow the generation of an output printer listing containing a maximum of compilation information. The action of assigning the state SET to the TRACE option sets the options CODE, DEBUG, LIST, and STACK. Once set, the TRACE option cannot be reset; however, any or all of the four TRACE sets may be reset individually.

- **VECTORMODE (Reset)**

This option is discussed in chapter 20.

COMPILER OPTIONS

- **VOID (Reset)**

The VOID compiler option is provided to allow the limiting of compiler source input for specific portions of compiler input files. When the state of VOID is SET, all input from files CARD and TAPE (except compiler control cards in the CARD file) will be ignored by the compiler. The ignored input will be neither listed nor included in the updated symbolic file regardless of the states of LIST and NEW. The VOID option may be reset (once it is set) only by a compiler control card in the CARD file.

- **VOIDT (Reset)**

The VOIDT compiler option is provided to allow the limiting of compiler source input for specific portions of the secondary compiler input file. When the state of VOIDT is SET, only primary input (i.e., input from file CARD) will be compiled; input from the secondary file TAPE (including compiler control cards) will be ignored until VOIDT is assigned the state RESET. The ignored input will be neither listed nor included in the updated symbolic file regardless of the states of LIST and NEW. The state of VOIDT is ignored by the compiler unless the option MERGE is set since secondary input is not accepted unless MERGE is set. See also the OMIT option.

- **XREF (Reset)**

The XREF compiler option causes a cross reference of all identifiers used in a compiled program to be output to the LINE file. The LIST compiler option need not be set to generate a cross reference listing. The identifiers are arranged according to the EBCDIC collating sequence with lists of sequence numbers of card images on which the identifier appears.

XREF should be set prior to the first source statement; once set, it may not be reset.

- **$ (Reset)**

The $ compiler option causes all compiler statements to be included in the LINE output file (listing) if the LIST option is also set.

FORTRAN COMPILER GENERATION OPTIONS

Several different versions of the standard FORTRAN compiler may be generated by setting or resetting the following user options when the compile is created from its ALGOL source. Each option is followed by its default setting.

- **BIGOPTJOB (Reset)**

This option facilitates the optimization of FORTRAN programs that are larger than the standard compiler handles. If set, BIGOPTJOB increases the memory requirements of the FORTRAN compiler. This option may be set only if OPTLSS1 is reset.

- **FASTGRAPH (Reset, but must be set if OPTLSS1 is reset)**

When set, this option causes graphs to be printed faster by ignoring end-of-page sensing on the line printer. When the option is reset, end-of-page sensing occurs and the graphs will not be drawn across a fanfold in the paper. (See chapter 20, OPTIMIZATION AND VECTORMODE.)

- **GETFIRSTDOLLARCARDFROMSPO (Reset)**

When this option is set, the compiler will require that the first compiler option card (dollar card) be input through the operator's supervisory console. This should normally be reset.

- **MODELII (Reset)**

If this option is set, the compiler will emit code to perform indexable value calls on arrays. This code is not emitted when the option is reset. The option should not be set unless the hardware feature is available (i.e., R.I.N. 0326-239 has been installed).

- **MULTI (Set)**

This option allows the compiler to optionally operate in batch mode. Batch facilities are not available if this option is reset. See the discussion of batch facilities in chapter 18.

- **OPTLSS1 (Reset)**

When this option is set, the FORTRAN compiler control option OPT may only be set to values less than 1. OPT values greater than zero are valid if OPTLSS1 is reset. A smaller compiler will result if OPTLSS1 is set.

COMPILER OPTIONS

- **SETOWNARRAYS (Reset)**

When set, this option causes the compiler to set compiler control option OWNARRAYS by default. OWNARRAYS will default to reset if SETOWNARRAYS is reset. When OWNARRAYS is set, arrays in subprograms are initialized to zero only when the subprogram is first called. On subsequent calls the arrays will retain their previous values, reducing execution time.

- **SINGLEBYDEFAULT (Set)**

When this option is set, compiler control option SINGLE will be set by default, causing single spaced FORTRAN listings. When SINGLEBYDEFAULT is reset, the listings will be double spaced by default.

- **VECTORMODEISALLOWED (Reset)**

When this option is set, the compiler may emit vector mode code under the proper conditions. When this option is reset, no vector mode code is created. (Refer to chapter 20 for discussion of vector mode.)

CHAPTER 20

OPTIMIZATION AND VECTORMODE

Optimization and Vectormode are two important facilities that in many cases will allow the compiler to generate faster-running object programs.

OPTIMIZATION

The parameter compiler option OPT may be used to alter the object code generated for a program by altering the manner in which the compile handles individual or groups of source input statements. The proper format of the OPT compiler control statement is:

OPT n or OPT = n

where n is the option argument which may be one of the integer constants -1, 0, or 1. The default value is 0. The value of OPT may be changed at any time. However, a change to or from the value 1 takes effect only at the beginning of a program unit. A change from 0 or -1 to the other takes effect with the next executable statement.

OPT = 0

When the value 0 is associated with OPT, the FORTRAN compiler performs local optimization within statements to take advantage of special hardware features. In addition, floating point constants with zero fractional parts are not normalized, that is, they are stored as integers. When this OPT value is used, the compiler enforces the rule that operands used in arithmetic expressions may be evaluated in any order.

OPT = -1

When the value -1 is associzted with OPT, compiler action is similar to that which occurs when the OPT value is zero, except that floating point integer constants are normalized and operands are evaluated from left to right so that any side effects will occur in the order written.

In order to illustrate the difference between OPT = 0 and OPT = -1, the following examples are provided. In the first example, assume that F is a one-parameter function sub-program which alters the value of its parameter when

the function is invoked and that A is a one-dimensional array. The first example consists of the input source statement:

```
A(I) = F(I) + A(I) + I
```

If this statement is evaluated from left to right, the value of I changes from the value it holds when it is first used as a subscript to the value it holds after the function F is invoked. This statement will be evaluated in this manner if the value associated with OPT is -1 or if the compilation is performed by a II.1 or earlier version FORTRAN compiler. With a value of 0 assigned to OPT, the subscript is evaluated only once (i.e., the value held by I at the time the execution of the statement is initiated) as the compiler supposes the user intended. The need to invoke F, however, will alter the value of I as employed in the last term of the expression.

A second example of the difference between OPT = 0 and OPT = -1 concerns a two-dimensional array B and the function F and variable I as presented in the previous example. The example statement is:

```
X= B(I,F(I))
```

The value associated with the OPT option at the time this statement is executed will determine which element of the B array has its value assigned to the variable X. When OPT=0, the I subscript will be referenced after the invocation of F and thus hold the value assigned to it by the execution of F, since F also is employed as a subscript expression. When OPT=-1, the two subscripts will be evaluated from left to right; thus, I will hold the value it contains before F is referenced. This latter procedure is the manner in which II.1 and earlier versions of the FORTRAN compiler evaluated such expressions.

Normally, (i.e., when side effects are not employed), the difference between OPT = 0 and OPT = -1 will have no effect on program results; however, the use of the OPT value of 0 may result in better code generation.

OPT = 1

When the value 1 is assigned to OPT at the beginning of a program unit, much more powerful optimization is invoked. This value is only valid if the FORTRAN compiler was compiled with option OPTLSS1 reset.

A function subprogram which has a label as a formal parameter, or any program unit referencing such a function, is not eligible for optimization. The compiler will generate a syntax error for any ineligible program unit if OPT = 1.

Since OPT may be changed on a program unit basis, the presence and use of ineligible function subprograms does not prevent other program units from benefiting, perhaps significantly, from optimization.

Additional Compiler Control Card Considerations

The OPT option affects the way in which OWN variables are handled. If OPT = 1, then OWN array handling is improved, but simple variables declared with OWN set do not result in the best code. See chapter 19 for the discussion of compiler options OWN and OWNARRAYS.

The compiler option GRAPH has been added to the FORTRAN compiler for use with optimization. If OPT > 0, GRAPH is set, and LIST is set when the END statement is encountered in a program unit, the compiler will produce a rough flow chart of the (pre-optimized) program unit; this can assist programmers in making fuller use of the optimization features by showing complex flows of control that reduce the effectiveness of optimization.

If OPT < 1, vectormode code will not be emitted by the FORTRAN compiler. See the discussion of vectormode elsewhere in this chapter.

FORTRAN statements MONITOR and DUMP cannot be used with OPT > 0; however, TRACE, STATISTICS, PROGRAM DUMP, and DUMP STATISTICS may be used.

The following is an example of a compiler option card using OPT:

```
$  SET  OPT = 1                                                    00000001
```

Although the B 6700 and B 7700 are code compatible, some constructs optimal on one computer are not optimal on the other. If the compiler option B7700 is set, code favoring the B 7700 is emitted. This option is reset by default, in which case the code best suited for the B 6700 is emitted. This option only applies when OPT > 0.

NOTE

If a program unit becomes too large, the following message will appear:

PVALUENO TOO SMALL, INCREASE IT

This indicates that the limit of compiler "definition points" has been exceeded. To correct this situation, the user should either break up the program unit into two or more subunits, or he should recompile on a version of the FORTRAN compiler that has BIGOPTJOB set. If the same message appears when BIGOPTJOB

is set and the program unit cannot be broken up, optimization is not available for that program unit.

VECTOR MODE

Vector mode is a special hardware feature of some B 6700 systems which can provide significantly increased efficiency in the handling of arrays and loop control. This hardware may be used only under certain conditions described below, but serves as a powerful tool when the requirements are met. For a detailed description of vector mode mechanisms, see the B 6700 System Miscellanea, Form No. 5000367.

The FORTRAN compiler will generate vector mode code if all of the following conditions are met:

a. The FORTRAN compiler has been compiled (from its ALGOL source file) with the option OPTLSS1 reset and the option VECTORMODEISALLOWED set.
b. The compiler option OPT has been set to a value greater than zero for the affected program unit and the unit is eligible for that level of optimization (see the description of optimization).
c. The compiler option VECTORMODE has been set for the portion of the program to be optimized.
d. The program unit contains a loop that has all of the following characteristics:
    1. It is a DO loop or a loop constructed with IF statements that functions equivalently to a DO loop.
    2. The loop is entered only from the "statement" immediately preceding the loop; that is, the loop is always fallen into. No statement branches into the loop and no statement branches to the first statement of the loop except the actual looping and testing code.
    3. The incremented expression is a constant less than or equal to 3.
    4. The final and initial values are integers and are constant with respect to the loop; that is, if the initial and final values are not constants, then they must not be altered within the loop.
    5. The value of the control variable is not changed inside the loop except when it is implicitly incremented at the end of the loop.
    6. The control variable is not in COMMON storage.

7. No CALL, I/O statements, PAUSE, ZIP, or other file-handling statements appear in the loop.
8. No function references appear in the loop.
9. Intrinsic calls including ** are not used unless the result of the call is invariant with respect to the loop.
10. If arrays or COMMON elements are altered within the loop, or arrays are referenced within the loop using a subscript that changes with each iteration within the loop, then there are no more than three such usages. That does not mean a limit of three arrays:

```
   DO 10 I = 1, N
10 A(I) = A(I + 1) + A(I - 1) + B(N)
```

The above loop would generate vector mode code if the other conditions are met, because there are only three arrays being used with varying subscripts. If B (N) had been replaced by B (I) or A (I + 2), then no vector code would be generated. If the statement had been

```
   10 B(N) = A(I) + A(I + 1) + A(I + 3)
```

then no vector mode would be generated, because while only three array subscripts were varying, a fourth array was being altered in value.

11. An array with a common element being altered or being used with varying subscripts is termed a "vector" array.

A vector array is not referenced in a purely conditional manner. If it is, no vector code is generated.

```
   DO 10 I = 1, 10000
   10 IF (I.LT.50) A(I) = A(I) + 1
```

The above example would not generate vector code.

```
   DO 10 I = 1, 10000
   A(I) = A(I) + 1
   10 IF (I.LT.50) A(I) = A(I) + 1
```

The above example would generate vector code if the other conditions are met.

12. The partial derivative of each of the subscript expressions with respect to the control variable is an integer which is constant with respect to the loop.

    This essentially means that if the subscript is too complicated, vector mode code will not be emitted.

```
DO 10 K = 1, N
10 A(I, J) = B(I, K)* C(K,J) + A(I,J)
```

    The above will give vector code, if other conditions are met.

```
DO 10 K = 1, N
10 A(K*K) = B(K + K)
```

    The above will not cause vector code to be generated because of the subscript of A. The subscript of B is acceptable.

13. There are no non-LONG arrays.
14. There are no formal arrays unless LONG is set.
15. The loop is an innermost loop, i.e., it does not contain any other loops.

An attempt to execute vector mode code on a processor not equipped with the proper hardware will result in a program termination.

APPENDIX A

SAMPLE PROGRAM

An example of a card deck containing a typical, valid B 6700/B 7700 FORTRAN program is presented here and discussed in the following paragraphs. The symbol <I> denotes an invalid character (e.g., a 1-2-3 punch) in card column 1. The cards in the deck appear in the order listed here.

```
<I>COMPILE FORTRAN/EXAMPLE WITH FORTRAN; EBCDIC
 $SET LIST RESET SINGLE
 FILE  6=PLOT
       SUBROUTINE PLOT(YMAX,YMIN)
 C  SUBROUTINE SUBPROGRAM GENERATING A PRINTER PLOT OF THE 101 ELEMENTS
 C  OF THE ARRAY Y VS. A SCALE RANGING FROM 0 TO 100.
       COMMON /A/ Y(101)
       REAL LINE(101),MINUS
       DATA BLANK,PLUS,MINUS,BAR,PLT/1H ,1H+,1H-,1H:,1H*/
       WRITE(6,1)
1      FORMAT('1      PLOT'//)
       X=1
       DO 2 I=1,101
       LINE(I)=MINUS
2      IF(MOD(I-1,10).EQ.0) LINE(I)=PLUS
       WRITE(6,3) YMIN,YMAX
3      FORMAT(15X,E15.5,80X,E15.5)
       DY=(YMAX-YMIN)/100.
100    YY=Y(X)
       IY=IFIX((YY-YMIN)/DY+0.5) + 1
       IF(IY.LT.1.OR.IY.GT.101) GO TO 4
       LINE(IY)=PLT
4      WRITE(6,5)LINE
```

SAMPLE PROGRAM

```
 5      FORMAT(20X,101A1)
        IF(MOD(LINES,10).EQ.0)WRITE(6,8)X-1
 8      FORMAT('+',3X,E15.5)
        LINES=LINES+1
        IF(MOD(LINES,10).EQ.0) GO TO 9
        DO 6 I=1,101
        LINE(I)=BLANK
 6      IF(MOD(I-1,10).EQ.0) LINE(I)=BAR
 7      X=X+1
        IF(X.GT.101) RETURN
        GO TO 100
 9      DO 310 I=1,101
        LINE(I)=MINUS
 310    IF(MOD(I-1,10).EQ.0) LINE(I)=PLUS
        GO TO 7
        END
C  START OF MAIN PROGRAM
$SET  SEQ
        DIMENSION X(101)
        COMMON/A/X
        DO 1 I=0,100
        X(I+1)=SIN(I/6.0)
 1      WRITE(6,2) I,X(I+1)
        CALL PLOT (1,-1)
C   THE MAXIMUM RANGE OF THE FUNCTION TO BE PLOTTED IS 1 TO -1
 2      FORMAT (I4,3X,F7.2)
        STOP
        END
<I>END OF EXAMPLE DECK
```

This sample deck is an example of a compile-and-go FORTRAN job. The end result of the execution of this program is a line printer plot and listing of values of the function:

$$y = \sin\left(\frac{x}{6}\right)$$

for x= 0,1,2,3,...,100. The plot produced contains grid lines and axis values, and each function value is plotted on one line of print beginning with a line for x=0 and terminating with a line for x=100. The program which produces

this output is constructed from two program units: a main program and a subroutine named PLOT. The deck containing these program units is described here.

The first card of the sample deck contains a COMPILE system control statement and an EBCDIC system control statement. This card causes a compile-and-go FORTRAN compilation to be initiated with the following EBCDIC-coded cards being used as source input for the compilation. The TITLE of the code file which is to be generated by the compiler and executed is FORTRAN/EXAMPLE. (This code file is not retained after execution.)

The first card in the compiler source file is a file declaration card which specifies that the program file referenced by the file designator 6 (and identified by the internal file name FILE6) be assigned the TITLE PLOT. (See appendix B.) The file attributes KIND (i.e., UNIT), MAXRECSIZE (i.e., RECORD), and BUFFERS (i.e., BUFFER) are allowed to retain their default values for this file of PRINTER, 22 words, and 2, respectively. The TITLE PLOT will appear in the first header of this print file.

The following card is a compiler control card which assigns the state SET to the compiler option LIST, causing the printer to produce a listing of the source program.

The following source input cards contain the subprogram PLOT. This subroutine generates the plot and has two formal arguments, YMAX and YMIN which specify the maximum range of the values to be plotted. The values to be plotted are transmitted to the subroutine via the common block labeled A to the array Y. The values contained in this 101-element array are plotted on the plot grid using the character "*". The plot grid is constructed line by line via the LINE array. The PLOT program unit illustrates the use of comment cards, the DATA statement, proper strings in FORMAT specifications, the MOD intrinsic, statement labels, etc., and contains three DO loops. The subroutine is terminated by the END statement which appears two statements after the IF statement with the label 310.

The subroutine is immediately followed by a comment card and a compiler control card which assigns the state SET to the SEQ compiler option. This control card causes the images of the following source input cards to bear sequence numbers when placed in the printer listing, beginning with the sequence number 00001000 and continuing in increments of 1000. The source cards following the subroutine constitute the main program.

SAMPLE PROGRAM

This main program places into the 101-element array X the 101 values of the sine function obtained by varying the value of the variable portion of the argument of the function in integer increments from 0 to 100. These values are transmitted to the PLOT array Y via the common block with the label A and are listed by the main program as they are calculated. The main program calls the PLOT subroutine with the actual parameters 1 and -1, which correspond to the formal parameters YMAX and YMIN, respectively.

The END system control card which terminates the compilation deck immediately follows the END statement which terminates the main program.

APPENDIX B

FILE DECLARATION CARDS

Default specifications are associated with each valid file designator and internal file name which are assigned to the program file referenced by the designator. A special, optional source statement type is provided which allows the user to assign non-default specifications to selected program files. These source statements are not part of the FORTRAN source language; when used, these file declaration statements must precede all FORTRAN source statements and serve to indicate to the compiler desired values of attributes of the specified program files. Attributes which do not have their values explicitly specified will retain their default values. The user is referred to the Input Output Subsystem Manual, Form No. 5000185, for a discussion of file attributes. Program files are discussed in chapter 12.

File declaration cards may also be used to inform the compiler of file designators which do not explicitly appear in the program but which may be referenced during program execution by an arithmetic expression of variable value.

A file declaration card is denoted by the appearance of the word FILE in columns 1 through 4 of the card. Columns 5 and 6 must be blank, with the remainder of the card being structured similar to a FORTRAN source input card; thus, file information is contained in card columns 7 through 72, and sequence numbers may be contained in columns 73 through 80 for merging purposes. These rules apply even if either of the following compiler options is set: FREE or FREETAPE. Also, as many continuation cards as desired may be used to extend the file declaration statement; the leftmost character on such a continuation card is a non-zero, non-blank character in card column 6, with the remaining columns being formatted in the same manner as the card on which the statement begins.

FILE DECLARATION CARDS

Two forms of the file declaration card are available. The first form resembles system control cards that affect file attributes. The file number, an unsigned integer from 1 to 99, is followed by a list of attribute assignments clauses separated by commas and enclosed in parentheses. Each clause is of the form fa=v, where fa is a valid file attribute identifier as listed in the Input/Output Subsystem Information Manual, Form No. 5000185, and v is an expression or mnemonic whose type and value are valid for that attribute identifier. This form of the file declaration card is not valid if compiler option B5500 or B5700 is set.

Any pointer valued attribute, such as TITLE, is set using an EBCDIC string; i.e., the compiler options ASCII, BCD and BCL must be reset when the source statement is encountered by the compiler. If the last character is not a period, the system will append a period to the end of the string for FILE statements; for OPEN and CHANGE statements, the user must supply the terminating period.

Example:

```
FILE 16 (TITLE="(USERME)MY/FYLE.",MAXRECSIZE=10,KIND=DISK)
```

Any attribute whose type is logical may be set by use of the attribute alone.

Example:

```
FILE 27 (TITLE="MINE",KIND=DISKPACK,CYLINDERMODE)
```

CYLINDERMODE, a logical attribute, will be set to TRUE.

In the second form, each file declaration statement consists of 1 to 13 file information clauses which are placed in free-field manner in columns 7 through 72 of the card(s) bearing the statement. Each such clause assigns a value to a file attribute.

The first file information clause is required and must be of the following form:

```
n = t
```

where n is either an unsigned integer file designator from 1 to 99, the literal READER, the literal PRINTER or PRINT, or the literal PUNCH, and t is the string of characters which are to be assigned as the value of the TITLE attribute of the designated program file. The following replacements for the item n produce the indicated effects:

a. When n is an unsigned integer constant, the present file declaration card pertains to the file referenced by all I/O statements appearing in the program being compiled which employ a file designator which assumes the value of this integer constant. Assuming FILE5, FILE6, and FILE7 are card reader, line printer, and card punch files, respectively, the following statements hold: if n is 5, then the file affected is also that which is referenced by input statements of the forms READ f or READ f,m unless a file declaration card appears for which n is PRINT or READER. If n is 6, then the file effected is also that which is referenced by output statements of the form: PRINT f or PRINT f,m unless a file declaration card appears for which n is PRINTER. If n is 7, then the file effected is also that which is referenced by output statements of the form: PUNCH f or PUNCH f,m unless a file declaration card appears for which n is PUNCH.

b. When n is READER, the present file declaration card pertains to the file referenced by all input statements appearing in the program being compiled of the form: READ f or READ f,m. The appearance of such a file declaration card results in the existence of a program file with the internal name READER. Hence, input statements of the form READ(5,f) or READ(5,f)m will reference the program file with the internal name FILE5, while other types of READ statements (i.e., those without file designators) will reference the file with the internal name READER rather than FILE5, which is the default condition when no file declaration card with an n of READER appears.

c. When n is PRINTER or PRINT, the present file declaration card pertains to the file referenced by all output statements appearing in the program being compiled of the form: PRINT f or PRINT f,m. The appearance of such a file declaration card results in the existence of a program file with the internal name PRINTER. Hence, output statements of the form WRITE(6,f) or WRITE(6,f)m will reference the program file with the internal name FILE6, while the PRINT statement will reference the file PRINTER rather than FILE6, which is the default file referenced by PRINT statements when no file declaration card with an n of PRINTER appears.

d. When n is PUNCH, the present file declaration card pertains to the file referenced by all output statements appearing in the program being compiled of the form PUNCH f or PUNCH f,m. The appearance of such a file declaration card results in the existence of a program file with the internal name PUNCH. Hence, output statements of the form WRITE(7,f) or WRITE(7,f)m will reference the program file with the internal name FILE7, while PUNCH statements will reference the file PUNCH rather than FILE7, which is the default file referenced by PUNCH statements when no file declaration card with an n of PUNCH appears.

NOTE

The files declared by file declaration cards bearing the designators READER, PRINTER (or PRINT), or PUNCH may be referenced only by the I/O statements READ (w/o file designator), PRINT, and PUNCH, respectively.

The item t may assume one of the following forms:

a. This item may be a series of 1 to 14 identifiers separated by character / and optionally delimited by blanks. This series of identifiers constitutes the directory title of the file to be associated with the specified internal file name; this title will be assigned to the TITLE attribute when the file is created or is the directory title of the existent file to be accessed. Each identifier in the file TITLE consists of 1 to 17 EBCDIC characters. The identifier may have one of two forms: it may begin with a letter character which is followed by 0 to 16 letter or digit characters (including the character $), or it may contain any valid EBCDIC characters including special characters (i.e., #, @, %, /, etc.). If the identifier contains any special characters, then the identifier must be delimited by apostrophes (') for EBCDIC-coded input or quotes (") for BCL- and BCD-coded input; such delimiters are not considered as part of the file TITLE.

Valid examples of this form of t are (for EBCDIC-coded input):

```
A9
A/BCDEFGHISTRQZXCVY
'A@BCD&F'
G678/'R&S'/ DATAFILE /'END-OF-TABULATION'
```

b. Alternately, t may assume a form which reflects the implementation of file security on this system. (See the System Miscellanea document.) A file TITLE may be used to replace t which is constructed as described in (a) with the addition that the series of identifiers may be preceded by the character * or the following item:

```
(u)
```

where u is a usercode which is constructed in the same manner as each identifier in the TITLE. This identifier may not be USERCODE. The * and (u) forms of t are for use with jobs run under a usercode. Under such conditions, every TITLE specified as in (a) will have the usercode automatically added to it as the leftmost identifier. For example, for a job run under the usercode USERX and employing a file declaration item t of A/B, the file actually referenced will be (USERX)A/B. An * preceding the TITLE indicates that no such addition is to occur, and a (u) item preceding the TITLE allows the user to specify the usercode to be added.

Valid examples of this form of t are (for EBCDIC-coded input):

```
* A9
(USERX) A/BCDEFGHISTROZXCVY
('USER-THREE') 'A@BCD&F'
* 'A/B'/CDE99
```

The n=t file information clause is the only required portion of a file declaration statement, aside from the initial "FILE" literal. Hence, the following are valid examples of file declaration cards for EBCDIC-coded input:

```
FILE  6=OUTPUT/'INFO-FILE'/ONE                                  00000001
FILE  PUNCH = FILESEVEN
```

One to twelve unique file information clauses (separated by commas) may follow the initial clause just described. These clauses are listed here and discussed in the following paragraphs. These clauses may appear on a file declaration card in any order.

FILE DECLARATION CARDS

```
UNIT = h
AREA = a or AREA = j * k
BLOCKING = b
BUFFER = c
CYLINDERMODE
INTERCHANGE
LINKWORD
RANDOM (should not appear if the clause SERIAL appears)
RECORD = r
SAVE = s
SERIAL (should not appear if the clause RANDOM appears)
SINGLEPACK
UNLABELED
```

The valid file information clauses are described as follows. Each clause is discussed in alphabetical order except UNIT, which is discussed first due to its importance.

- **UNIT = h**

This file information clause is provided to specify the physical device used by the file. The device employed is indicated by the item h, which may be one of the following hardware unit names:

PRINT	(007)
PRINTER	(007)
PRINT,BD	(071)
PRINTER,BD	(071)
PRINT,BT	(135)
PRINTER,BT	(135)
PRINT,BDBT	(199)
PRINTER,BDBT	(199)
READER	(009)
PUNCH	(011)
PUNCH,BD	(075)
PUNCH,BT	(139)
PUNCH,BDBT	(203)
DISK	(001)
TAPE	(045)
TAPE7	(013)
TAPE9	(014)
PETAPE	(015)
REMOTE	(003)
DISKPACK	(017)

In the list just presented, the integer value in parentheses following the unit name is the value of the FILEKIND attribute associated with that unit. The unit names PRINT and PRINTER indicate that the file is an output printer file. PRINT,BD and PRINTER,BD indicate that the file is a printer backup disk file. PRINT,BT and PRINTER,BT indicate that the file is a printer backup tape file. PRINT,BDBT and PRINTER,BDBT indicate that the file is a printer backup tape file unless no tape is available, in which case the file is a printer backup disk file. READER indicates that the file is an input card file. PUNCH indicates that the file is an output card punch file. PUNCH,BD and PUNCH,BT indicate that the file is a punch backup disk file or punch backup tape file, respectively. PUNCH,BDBT indicates that the file is a punch backup tape file unless no tape is available, in which case the file is a punch backup disk file. DISK indicates that the file is a disk file. TAPE indicates that the file is a magnetic tape file (i.e., any available tape). TAPE7, TAPE9, and PETAPE indicate that the file is assigned to 7-track, 9-track, or phase-encoded magnetic tape, respectively. REMOTE indicates that the file is assigned to a remote terminal. DISKPACK indicates that the file is contained in a disk-pack. The file attribute affected by this clause is the KIND attribute.

A valid example of this clause is:

```
UNIT = PRINT,BD
```

- **AREA = a or AREA = j * k**

This file information clause is provided for use with disk files (UNIT=DISK or UNIT=DISKPACK). By use of this clause, a FORTRAN source program may specify the size and number of disk areas that a disk file is to use. The AREA clause implies that the file is an output file and is required for such files. The item a is an unsigned integer constant which is related to the size of the disk file areas (rows) in the following manner. The value of a is the total number of logical records assigned to the disk file and is divided evenly among 20 areas. The value of a will be automatically increased to the next integer multiple of 20 if necessary. This value should be chosen such that a/20/b is an integer value, where b is the blocking (i.e., the number of physical records per disk segment).

For the alternate form of the AREA clause, j is the number of logical records in each disk area and k is the number ($1 \leq k \leq 1023$) of disk areas assigned to the file, and each is an unsigned integer constant. The value of j should be

chosen so that j/b is an integer value, where b is the file blocking. Unpredictable end-of-file conditions occur if j is not selected as described. The file attributes affected by this clause are AREAS and AREASIZE.

The following are valid examples of the AREA clause:

```
AREA=100
AREA=1000*10
```

The first example specifies that the file contain a total of 100 records, distributed such that each of the 20 default disk areas contains 5 logical records. The second example specifies that the file be assigned 10 areas, each of which contains 1000 logical records. Thus, the AREAS attribute is assigned the value 10, and the AREASIZE attribute is assigned the value 1000.

NOTE

The a in the first form of this clause may be thought of as the product of a default k of 20 in the second form and a chosen areasize j (i.e., (a+19) DIV 20).

- **BLOCKING = b**

This file information clause is provided to specify the number of logical records to be assigned to each physical record (for disk files, the size of a physical record should be an integer multiple of the 30-word segment). Item b is an unsigned non-zero integer constant which specifies the desired blocking. The default blocking is 1 (i.e., unblocked).

- **BUFFER = c**

This file information clause is provided to specify the number of buffers to be used with the file. Item c is an unsigned non-zero integer constant which denotes the number of buffers to be employed. The default number of buffers is 2 for all files. The size of each buffer is identical to the size of a physical file record, which may be found by taking the product of the assigned blocking and the logical record size. No buffer may be longer than 65,535. In general, job times will be diminished by assigning more than two buffers to jobs performing a great deal of I/O operations. This clause effects the value of the BUFFERS file attribute.

The following is a valid example of the BUFFER clause:

```
BUFFER = 6
```

- **CYLINDERMODE**

The appearance of this file information clause assigns the value TRUE to the Boolean file attribute CYLINDERMODE; the default value of this attribute is FALSE. This attribute is meaningful for disk-pack files (UNIT=DISKPACK) only. When TRUE, this attribute provides for assignment of file space by disk-pack cylinders such that each file area (row) lies within the bounds of a single cylinder. When the value of this attribute is FALSE, file space assignment is made without regard for cylinder bounds but rather on a space available basis. The size of each file area must not be greater than the cylinder size when the CYLINDERMODE clause appears or an error condition will exist.

The correct form of the CYLINDERMODE clause is:

```
CYLINDERMODE
```

- **INTERCHANGE**

The appearance of this file information clause assigns the value TRUE to the Boolean file attribute INTERCHANGE; the default value of this attribute is FALSE. This attribute is meaningful for disk-pack files (UNIT=DISKPACK) only. When TRUE, this attribute indicates that the file is stored or is to be stored on a Burroughs Interchange pack.

- **LINKWORD**

The LINKWORD file information clause is intended for use with disk and tape files which are accessed via unformatted-I/O statements. This clause may be used to indicate that the length of a logical file record may be greater than the length of a physical record. The appearance of this clause assigns the value 6 to the file attribute FILETYPE.

The correct form of the LINKWORD clause is:

```
LINKWORD
```

- **RANDOM**

The RANDOM file information clause is provided for use solely as a documentation aid with disk files. Since there is no distinction between "random" and "serial" disk files on this system, the presence of the RANDOM and SERIAL clauses are ignored. The RANDOM clause may be used to document the fact that the associated disk file is to be used as if it were a "random-access" file.

The RANDOM file clause has the form:

```
RANDOM
```

FILE DECLARATION CARDS

- **RECORD = r**

The RECORD file information clause is provided to allow the specification of the size of each logical record in the file. Item r is an unsigned integer constant which may have a value in the range 1 to 65,535 and which specifies the number of data words contained in each logical record. The default record sizes are 14 words for reader and punch files, and 22 words for all other files. Care should be taken that buffer size, which is the product of blocking and record size, does not exceed 65,535.

The RECORD clause affects the MAXRECSIZE file attribute. This example record clause:

```
RECORD = 30
```

assigns the value 30 to the MAXRECSIZE attribute of the file.

- **SAVE = s**

This file information clause is provided to allow the user to specify the length of time an output disk file is to remain in the file directory. Item s is an unsigned integer constant, the value of which may not exceed 999. This item specifies the number of days the file is to be retained. The SAVE clause effects the value of the SAVEFACTOR file attribute. This sample SAVE clause:

```
SAVE = 30
```

assigns the value 30 to the SAVEFACTOR attribute of the file.

- **SERIAL**

The SERIAL file information clause is provided for use solely as a documentation aid with disk files. Since there is no distinction between "serial" and "random" disk files on this system, the presence of the SERIAL and RANDOM clauses are ignored. The SERIAL clause may be used to document the fact that the associated disk file is to be used as if it were strictly a "serial-access" file. A warning is issued when used.

The SERIAL file clause has the form:

```
SERIAL
```

- **SINGLEPACK**

The appearance of this file information clause assigns the value TRUE to the Boolean file attribute SINGLEPACK; the default value of this attribute is FALSE. This attribute is meaningful for disk-pack files (UNIT = DISKPACK) only. When TRUE, this attribute specifies that the areas of an output file

are to be assigned to a single pack. When the value of this attribute is FALSE, file areas will be uniformly distributed among all packs of the pack set which is present when the file is open. The left-most identifier of the file TITLE (i.e., the multi-file ID) must match the name of the pack set. Thus, the file PACKFILE/ONE requires a pack or packset PACKFILE.

The correct form of the SINGLEPACK clause is:

SINGLEPACK

- **UNLABELED**

This file information clause is provided to specify that an output file is to have an omitted label. The presence of this clause assigns the value 1 (OMITTED) to the file attribute LABELTYPE. The default specification (when this clause is missing) is a LABELTYPE value of 0 (STANDARD), except for REMOTE files which are unlabeled by default.

The correct form of the UNLABELED clause is:

UNLABELED

EXAMPLES:

The following are examples of valid file declaration cards which may represent either physical punched cards or card images in another type of source input file.

a. The first example consists of this source card:

```
FILE  PRINTER=ANALYSIS/L198, UNIT=PRINT,BD,  BUFFER=6      00000010
```

This file declaration card assigns the TITLE ANALYSIS/L198 to the program file with the internal name printer, designates this file as a printer backup disk file, and assigns it six buffers. The card has a sequence number of 10.

b. The second example consists of these source cards:

```
FILE  4=PACKSET2/RESULTS, UNIT=DISKPACK, SAVE=20
     1,CYLINDERMODE
     2,INTERCHANGE
```

This file declaration card (with continuation cards) assigns the TITLE PACKSET2/RESULTS to the program file denoted by the file designator 4, designates the file as a disk-pack file to be retained for 20 days, and indicates that file areas are to be assigned by cylinders to Burroughs Interchange packs.

c. The third example consists of these source cards:

```
FILE  5=                                                    00000001
      1* CARDDATA/'HALF-SET', UNIT=DISK, RECORD=15           00000002
```

This file declaration card (with continuation card) equates the program file denoted by the file designator 5 to the disk file with the TITLE CARDDATA/HALF-SET. The * indicates that the file TITLE does not have an additional left-most identifier of the usercode under which the job is being run. The file has 15-word physical records.

As an addition to this example, assume this file declaration card is preceded by this file system control card:

```
<I>FILE FILE5(TITLE=*CARDDATA/"HALF-SET", MAXRECSIZE=13)
```

In the event of conflicts between this system control card and the file declaration source card, the control card overrides the source card. Both the system control statement and the file declaration statement in this example apply to the program file with the internal name FILE5. The control card indicates that the actual size of file records is 13 words rather than 15. The TITLE specification need not have appeared on this card.

SEPARATE COMPILATION CONSIDERATIONS

Each program unit in a compilation deck may be immediately preceded by file declaration statements if the state of the compiler option SEPARATE is SET for the compilation. At compilation time, file declaration cards apply to all subsequent program units. However, a file may be redeclared preceding a later program unit. In this case, the new file declaration will apply to all subsequent program units that are compiled until the file is redefined.

At execution time, the file declaration for a given file designator that is bound to the host program first will have precedence. File declarations for main program hosts will apply to all subprograms that are bound to that main program. Such file declarations may be incurred via file declaration cards preceding the main program when that main program was compiled, or via default declarations resulting from the appearance of file designators in the main program.

The following sample compilation deck illustrates the use of file declaration statements with separate compilations.

```
<I> COMPILE SEP/PROC WITH FORTRAN LIBRARY; EBCDIC
 $SET SEPARATE SINGLE
 FILE  10=ABC, UNIT=TAPE
       COMMON A(10)
       DATA A/10*3/
       WRITE(10) A
       CALL SUB
       Y=F(1)
       STOP
       END
 FILE  10=ARP,UNIT=DISK,AREA=100
 FILE  7=ERB,UNIT=DISK,AREA=100
 FILE  5=EHU,UNIT=TAPE
       FUNCTION F(X)
       COMMON A(10)
       WRITE(7) A
       WRITE(10)A
       WRITE(5) A
       F=X
       RETURN
       END
 FILE  10=ETX/FILE10,UNIT=DISKPACK,SINGLEPACK
 FILE  5=FILE5/AMK,UNIT=DISK,AREA=1000
       SUBROUTINE SUB
       COMMON A(10)
       WRITE(5) A
       WRITE(10)A
       RETURN
       END
<I>END
```

When this main program and two subprograms are bound and executed, file assignments will be made as follows.

a. All writes on FILE10 (the file referenced by the designator 10) will be on the program file declared before the main program.

b. All writes on FILE5 (referenced by designator 5) will be on the disk file declared preceding SUB, since SUB is referenced before F in the main program and hence is bound in first.

c. All writes on FILE7 (referenced by designator 7) will be on the disk file declared preceding F since this file is neither referenced in the main program nor redeclared preceding SUB.

FILE DECLARATION CARDS

DEFAULT FILE SPECIFICATIONS

Each file designator is associated by default with a certain type of program file. The standard FORTRAN associations are made between the designators 5, 6, and 7 and card reader, line printer, and card punch files, respectively. Table B-1 lists default file specifications by internal file name.

Table B-1. Default Program File Specifications

INTERNAL FILE NAME	DEFAULT FILE ATTRIBUTE VALUES				
	TITLE	MAXRECSIZE	BLOCKSIZE	BUFFERS	KIND
FILE1 thru FILE4	FILE1 thru FILE4	22	22 (unblocked)	2	Tape
FILE5	FILE5	14	14 (unblocked)	2	Card Reader
READER *	specified on FILE card	14	14	2	Card Reader
FILE6	FILE6	22	22 (unblocked)	2	Line Printer
PRINTER *	specified on FILE card	22	22 (unblocked)	2	Line Printer
FILE7	FILE7	14	14 (unblocked)	2	Card Punch
PUNCH *	specified on FILE card	14	14 (unblocked)	2	Card Punch
FILE8 thru FILE99	FILE8 thru FILE99	22	22 (unblocked)	2	Tape

*These files exist only if file declaration cards appear at the start of the source deck bearing "READER", "PRINTER" (or "PRINT"), and/or "PUNCH" as the effected file designators.

APPENDIX C

FORTRAN COMPILER ERROR MESSAGES

The following is a list of error messages produced by the FORTRAN compiler. Beneath each error message is a list of the compiler procedure (or procedures) in which the message may be invoked.

NO.	ERROR MESSAGE AND COMPILER PROCEDURES
0	SYNTAX ERROR
	EMITADDRESS, EXPR2, IFS, IMPLICIT, SCANNER, SCANR, TRANSCLASS
1	MISSING OPERATOR OR ILLEGAL PUNCTUATION
	EXPR, TRANSCLASS
2	CONFLICTING COMMON AND/OR EQUIVALENCE ALLOCATION
	BUILDFIB, DIMENSION, FIXRING, SPACEFORCOMMON, SPACEFOREQUIVALENCE
3	MISSING RIGHT PARENTHESIS
	LOOKFORINTRINSIC, TRANSCLASS
4	ENTRY STATEMENT ILLEGAL IN MAIN OR BLOCK DATA
	ENTRY
5	MISSING END STATEMENT
	BLOCKDATA, Driver of the Compiler, FUNCTION, SUBROUTINE
6	ARITHMETIC EXPRESSION REQUIRED
	ASSIGNMENT, GOTOONE, GOTOS
7	SEQUENCE ERROR

FORTRAN COMPILER ERROR MESSAGES

NO.	ERROR MESSAGE AND COMPILER PROCEDURES
8	TOO MANY LEFT PARENTHESES MOVENUM
9	TOO MANY RIGHT PARENTHESES TRANSCLASS
10	FORMAL PARAMETER ALSO APPEARS IN COMMON DIMENSION
11	FORMAL PARAMETER ILLEGAL IN EQUIVALENCE EQUIVALENCE
12	STATEMENT ILLEGAL IN BLOCK DATA SUBPROGRAM EXPR, EXTERNALS, NAMEL, STMTFUN
13	INFO TABLE OVERFLOW - COMPILER LIMIT EXCEEDED PURGEINFO
14	IMPROPER DO NESTING CHECKLABELI, DATADO, STATEMENT
15	DO LABEL APPEARS BEFORE DO STATEMENT ASSIGNMENT
16	UNRECOGNIZED STATEMENT CHECKRESERVED, LABELR, PRESCANNER, PRESCANNER2, STATEMENT
17	ILLEGAL DO STATEMENT ASSIGNMENT, ASSIGNMENTSTATEMENTORDOLOOPONE
18	MISSING LABEL ON FORMAT STATEMENT - STMT IGNORED MOVENUM

NO.	ERROR MESSAGE AND COMPILER PROCEDURES
19	LABEL REFERENCED BUT NOT FOUND CLEANOUTINFO
20	MULTIPLE DEFINITION DECLARELABEL, NAMEL
21	ILLEGAL IDENTIFIER CLASS IN THIS CONTEXT DATADO, FCNREF, GETSPACE, NEED, SPACEFOREQUIVALENCE
22	PROGRAM UNIT TOO LARGE, INCREASE CODE.AREASIZE SEGMENT, WRITEITOUT NOTE To increase the AREASIZE of the compiler code file the following type of FILE MCP control card should be used: <I>FORTRAN FILE CODE(AREASIZE=200) The default code file AREASIZE is 180 records. The NEWSEGMENT compiler option may also be used.
23	NOT ENOUGH SUBSCRIPTS
24	TOO MANY SUBSCRIPTS ASSIGNMENTSTATEMENTORDOLOOPONE, SUBSCRIPTS, SUBSCRIPTSI
25	SUBPROGRAM PREVIOUSLY DEFINED FORMALPP
26	FORMAL PARAMETER MULTIPLY DEFINED FORMALPP
27	VARIABLE FILE MUST BE DECLARED WRAPUP
28	NUMBER OF PARAMETERS CONFLICTS WITH PREVIOUS USE PARAMETERS

FORTRAN COMPILER ERROR MESSAGES

NO.	ERROR MESSAGE AND COMPILER PROCEDURES
29	INCLUDE NESTED DEEPER THAN 5 LEVELS
30	THIS OPTION MUST PRECEDE SOURCE DECK SCANNER
31	IDENTIFIER TYPE PREVIOUSLY DEFINED DIMENSION
32	TOO FEW CONSTANTS IN DATA PART DATA
33	INCORRECT USE OF FILE IOCOMMAND, ZIPIT
34	IMPROPER FILE NAME ON INCLUDE CARD
35	ARRAY IDENTIFIER EXPECTED DATADO, SUBSCRIPTS, SUBSCRIPTSI
36	FILE NAME NEEDED ON INCLUDE CARD
37	NON-BLANK CHARACTERS DELETED FROM FREEFIELD CARD
38	INVALID SEQUENCE NO. SPECIFICATION ON $ INCLUDE
39	DECLARATION OCCURS AFTER FIRST REFERENCE DIMENSION, EQUIVALENCE, SPACEFOREQUIVALENCE
40	INCONSISTENT USE OF LABEL AS PARAMETER FORMALPP
41	NO. OF PARAMETERS DISAGREES WITH PREVIOUS REF. FORMALPP

NO.	ERROR MESSAGE AND COMPILER PROCEDURES
42	CONSTANT GREATER THAN MAXIMUM INTEGER BOUNDS, DATA
43	ERROR IN HOLLERITH LITERAL CHARACTER COUNT HOLLERITH, MOVENUM
44	ILLEGAL ACTUAL PARAMETER GETP, LOOKFORINTRINSIC, PARAMETERS, PRIMARY
45	COMMON BLOCK CONTAINS MORE THAN 65535 ELEMENTS SPACEFORCOMMON
46	INCLUDE FILE NOT ON DISK
47	EQUIVALENCE MAY NOT EXTEND COMMON BACKWARD SPACEFOREQUIVALENCE
48	ILLEGAL I/O LIST ELEMENT DATA, NAMEL, TRANSCLASS
49	LEFT PART MUST BE SIMPLE OR SUBSCRIPTED VARIABLE EMITSTORE
50	VARIABLE EXPECTED ASSIGNMENT, EXPR, PRIMARY, TRANSCLASS
51	ILLEGAL USE OF .OR. EXPR, MAKEEXPRESSION2
52	ILLEGAL USE OF .AND. EXPR, MAKEEXPRESSION2

FORTRAN COMPILER ERROR MESSAGES

NO.	ERROR MESSAGE AND COMPILER PROCEDURES
53	ILLEGAL USE OF .NOT. EXPR, EXPR1, MAKEEXPRESSION1
54	ILLEGAL USE OF RELATIONAL OPERATOR EXPR
55	ILLEGAL MIXED TYPES DATA, EXPR1, GETP, MAKEEXPRESSION1, MAKEEXPRESSION2
56	ILLEGAL EXPRESSION STRUCTURE EXPR, EXPR1
57	SIMPLE INTEGER VARIABLE REQUIRED ASSIGN, GOTOS
58	LABEL APPEARED IN ASSIGNED-GO-TO BUT NOT ASSIGN SEGMENT
59	ASSIGN STATEMENT(S) BUT NO ASSIGNED-GO-TO FOUND SEGMENT
60	FILE DECLARATIONS MUST PRECEDE SOURCE DECK LABELR
61	UNFORMATTED WRITE STATEMENT HAS NO LIST FORMATTEDIO, LOGRECWRITE
62	FORMAT REFERENCED BUT NOT DEFINED CLEANOUTINFO
63	ILLEGAL EXPONENT IN CONSTANT CHECKPERIOD

NO.	ERROR MESSAGE AND COMPILER PROCEDURES
64	ILLEGAL CONSTANT IN DATA OR INITIAL VALUE LIST DATA
65	ACTUAL PARAMETER MUST BE A LABEL PARAMETERS
66	ACTUAL PARAMETER MUST BE ARRAY IDENTIFIER FIXPARAMS, PARAMETERS
67	ACTUAL PARAMETER MUST BE EXPRESSION FIXPARAMS, PARAMETERS
68	ACTUAL PARAMETER MUST BE A (SUBSCRIPTED)VARIABLE PARAMETERS
69	PARAMETER MUST BE FUNCTION IDENTIFIER PARAMETERS
70	PARAMETER MUST BE SUBPROGRAM IDENTIFIER PARAMETERS
71	PARAMETER MUST BE SUBROUTINE IDENTIFIER PARAMETERS
72	PARAMETER MUST BE ARRAY IDENTIFIER OR EXPRESSION PARAMETERS
73	LABEL NOT ALLOWED ON THIS STATEMENT - IGNORED ASSIGNMENTSTATEMENTORDOLOOPONE, BLOCKDATA, DIMENSION, ENDS, ENTRY, EXTERNALS, FUNCTION, IMPLICIT, NAMEL, PRESCANNER, STMTFUN, SUBROUTINE

FORTRAN COMPILER ERROR MESSAGES

NO.	ERROR MESSAGE AND COMPILER PROCEDURES
74	ARRAYID MUST BE SUBSCRIPTED IN THIS CONTEXT ASSIGNMENT, DATADO, EXPR, PRIMARY
75	MORE THAN ONE MAIN PROGRAM SEGMENT
76	NUMBER OF PARAMETERS EXCEEDS 63 FORMALPP, PARAMETERS, STMTFUN
77	LEVEL OPTION MAY NOT BE RESET
78	FORMAT OR NAMELIST TOO LONG NAMEL
79	FORMAL PARAMETER MUST BE ARRAY IDENTIFIER FIXPARAMS
80	FORMAL PARAMETER MUST BE SIMPLE VARIABLE FIXPARAMS
81	FORMAL PARAMETER MUST BE FUNCTION IDENTIFIER FIXPARAMS
82	IDENTIFIER TRUNCATED TO 6 CHARACTERS ASSIGNMENT, ASSIGNMENTSTATEMENTORDOLOOPONE, SCANR
83	FORMAL PARAMETER MUST BE SUBPROGRAM IDENTIFIER FIXPARAMS, PUTSYL
84	DO OR IMPLIED DO INDEX MUST BE INTEGER OR REAL ASSIGNMENT

NO.	ERROR MESSAGE AND COMPILER PROCEDURES
85	ILLEGAL COMPLEX CONSTANT DATA, EXPR, PRIMARY
86	BOTH SIDES MUST BE LOGICAL EMITSTORE, TYPETRANSFER
87	CONSTANT EXCEEDS HARDWARE LIMITS CHECKPERIOD
88	PARAMETER TYPE CONFLICTS WITH PREVIOUS USE FIXPARAMS, LOOKFORINTRINSIC, PARAMETERS
89	COMPLEX EXPRESSION ILLEGAL IN IF STATEMENT IFONE, IFS
90	COMPLEX EXPRESSION ILLEGAL IN RELATIONAL EXPR
91	INVALID LITERAL FOR DISK AREA SPECIFICATION BUILDFIB
92	VARIABLE BOUND MUST BE FORMAL PARAMETER BOUNDS
93	ARRAY BOUND MUST HAVE INTEGER OR REAL TYPE BOUNDS
94	COMMA OR RIGHT PARENTHESIS EXPECTED ACTIONLABELS, BOUNDS, DATADO, IOCOMMAND, PRIMARY
95	ARRAY ALREADY DIMENSIONED BOUNDS

FORTRAN COMPILER ERROR MESSAGES

NO.	ERROR MESSAGE AND COMPILER PROCEDURES
96	ONLY FORMAL ARRAYS MAY HAVE VARIABLE BOUNDS BOUNDS
97	MISSING LEFT PARENTHESIS IN IMPLIED DO TRANSCLASS
98	SUBSCRIPT MUST BE INTEGER OR REAL DATAO, SIMPLEEXPRESSION
99	ARRAY SIZE CANNOT EXCEED 65535 ELEMENTS BOUNDS, FIXRING
100	LEVEL OPTION MUST PRECEDE PROGRAM UNIT SCANNER
101	THIS STATEMENT NOT ALLOWED IN LOGICAL IF ASSIGNMENT, ASSIGNMENTSTATEMENTORDOLOOPONE, DATA, DIMESION, ENDS, ENTRY, EQUIVALENCE, EXTERNALS, IFONE, IFS, IMPLICIT, MOVENUM, NAMEL, PRESCANNER2
102	REAL OR INTEGER TYPE REQUIRED ASSIGNMENT, DOASEEK, FILECONTROL, GOTOONE, GOTOS, IOCOMMAND, TRANSCLASS, ZIPIT
103	ARRAY BOUND INFORMATION REQUIRED DIMENSION
104	REPLACEMENT OPERATOR EXPECTED ACTIONLABELS, ASSIGNMENT, BUILDFIB, DATADO, STMTFUN
105	IDENTIFIER EXPECTED ASSIGN, BUILDFIB, COMMON, DATADO, DIMENSION, EQUIVALENCE, EXTERNALS, FORMALPP, NAMEL

NO.	ERROR MESSAGE AND COMPILER PROCEDURES
106	LEFT PARENTHESIS EXPECTED ACTIONLABELS, DOASEEK, EQUIVALENCE, FORMALPP, GOTOONE, GOTOS, IFONE, IFS, IMPLICIT, LOOKFORINTRINSIC, MOVENUM
107	ILLEGAL FORMAL PARAMETER FORMALPP, STMTFUN
108	RIGHT PARENTHESIS EXPECTED DATA, DATADO, DOASEEK, EQUIVALENCE, EXPR, FORMALPP, FORMATTEDIO, GETP, GOTOONE, GOTOS, IFONE, IFS, IOCOMMAND, PARAMETERS, PRIMARY, STMTFUN, TRANSCLASS
109	STATEMENT NUMBER EXPECTED ASSIGN, GOTOONE, GOTOS, IFONE, IFS
110	SLASH EXPECTED COMMON, DATA, FORMALPP, NAMEL
111	ENTRY STATEMENT CANNOT BEGIN PROGRAM UNIT ENTRY
112	ARRAY MUST BE DIMENSIONED BEFORE EQUIV.STMNT. BOUNDS, EQUIVALENCE
113	INTEGER CONSTANT EXPECTED DATA, DATADO, EQUIVALENCE, GETANUMBER, IMPLICIT, OPTIONALAST
114	COMMA EXPECTED BUILDFIB, DATA, DATADO, FORMATTEDIO, GETP, IFONE, IFS, LOOKFORINTRINSIC, TRANSCLASS
115	SLASH OR END OF STATEMENT EXPECTED NAMEL

FORTRAN COMPILER ERROR MESSAGES

NO.	ERROR MESSAGE AND COMPILER PROCEDURES
116	FORMAT, ARRAY, OR NAMELIST EXPECTED FORMATTEDIO
117	END OF STATEMENT EXPECTED PRESCANNER, PRESCANNER2, STATEMENT
118	I/O WITH NAMELIST CANNOT HAVE I/O LIST
119	COMMA OR END OF STATEMENT EXPECTED DATA, LOGRECWRITE
120	STRING TOO LONG: INCREASE MAXSTRING HOLLERITH, PARAMETERS, PAUSE, QUOTESTRING
121	MISSING QUOTE AT END OF STRING QUOTESTRING
122	ILLEGAL ARRAY BOUND BOUNDS
123	ILLEGAL FORTRAN USER INTRINSIC GETP, LOOKFORINTRINSIC
124	ILLEGAL LEX LEVEL SCANNER
125	INTEGER GEQ 1 EXPECTED ACTIONLABELS, DATA, DATADO
126	COMMA OR SLASH EXPECTED DATA

NO.	ERROR MESSAGE AND COMPILER PROCEDURES
127	ILLEGAL USER INTRINSIC PARAMETER PARAMETERS
128	MUST BE IMPLIED DO INDEX VARIABLE DATADO
129	D > W IN A SPECIFICATION OF THE FORM FW.D MOVENUM
130	UNSPECIFIED FIELD WIDTH MOVENUM
131	UNSPECIFIED SCALE FACTOR
132	ILLEGAL FORMAT CHARACTER MOVENUM
133	UNSPECIFIED DECIMAL FIELD MOVENUM
134	MUST SUBSCRIPT ALL ARRAYS IN ITS BOUNDS FOR DATA DATADO
135	ILLEGAL STATEMENT NUMBER ASSIGNMENT, EMITLABELDESC, FORMATTEDIO, LABELASSIGN, LABELBRANCH
136	UNDEFINED NAMELIST CLEANOUTINFO
137	MULTIPLY DEFINED ACTION LABELS ACTIONLABELS

FORTRAN COMPILER ERROR MESSAGES

NO.	ERROR MESSAGE AND COMPILER PROCEDURES
138	TOO MANY NESTED DOS ASSIGNMENT, ASSIGNMENTSTATEMENTORDOLOOPONE, PAUSE
139	STMT.FUNCTION ID AND EXPRESSION DISAGREE IN TYPE
140	ILLEGAL USE OF STATEMENT FUNCTION STMTFUN
141	UNRECOGNIZED CONSTRUCT ENTERLABEL, SCANR
142	STOP, RETURN, OR CALL EXIT REQUIRED IN SUBPGM ENDS
143	FORMAT NUMBER USED PREVIOUSLY AS LABEL FORMATTEDIO, MOVENUM
144	LABEL USED PREVIOUSLY AS FORMAT NUMBER EMITLABELDESC, ENTERLABEL, LABELASSIGN, LABELBRANCH
145	THIS COMMON BLOCK WILL BE SEGMENTED WRAPUP
146	DOUBLE OR COMPLEX REQUIRES EVEN OFFSET SPACEFORCOMMON, SPACEFOREQUIVALENCE
147	COMPILE-AND-GO INVALID WITH SEPARATE COMPILATION WRAPUP
148	VARIABLE OR ARRAY IDENTIFIER REQUIRED ACTIONLABELS

NO.	ERROR MESSAGE AND COMPILER PROCEDURES
149	COMMON ELEMENT INITIAL VALUE ILLEGAL IN SUBPRGM DATA, TRANSCLASS
150	MISSING SUBROUTINE OR FUNCTION WRAPUP
151	TOO MANY GLOBAL DECLARATIONS BREAKBLOCK, BUILDPCW, EMITLABELDESC, ENTERDATASTAK, FIXRING, FIXUPCALLS, FLUSHPOOL, FORMATSAVER, GETSPACE, INLINEFILE, INTERPHASE, LABELBRANCH, LOOKFORINTRINSIC, PARAMETERS, PAUSE, SEGMENT, SEGMENTSTART, STMTFUN, STOP, TRANSCLASS, WRAPUP, WRITESEPFPBS NOTE Overflow of D[1] or D[2] stack array occurred.
152	TOO MANY LOCAL DECLARATIONS EMITNAMCLIT, EMITVALCLIT NOTE Overflow of D[3] or D[4] stack array occurred.
153	EQUIVALENCE TABLE OVERFLOW - INCREASE LSTMAX EQUIVALENCESPACE, FORMALPP, PARAMETERS
154	MAX CHARS/WORD EXCEEDED BY STRING IN EXPRESSION EXPR, PRIMARY, SIMPLEEXPRESSION
155	PARTIAL WORD LENGTH SPECIFICATION IGNORED IMPLICIT, OPTIONALAST
156	ILLEGAL LETTER LIMIT PAIR IN IMPLICIT SPEC. IMPLICIT

FORTRAN COMPILER ERROR MESSAGES

NO.	ERROR MESSAGE AND COMPILER PROCEDURES
157	IMPLICIT TYPE CONFLICTS WITH PREVIOUS IMPLICIT IMPLICIT
158	IMAGINARY TRUNCATION OCCURS ON THIS REPLACEMENT EMITSTORE, TYPETRANSFER
159	ZERO IMAGINARY PART SUPPLIED ON THIS REPLACEMENT EMITSTORE
160	ERROR LIMIT EXCEEDED
161	ILLEGAL SYNTAX IN FIND STATEMENT DOASEEK
162	NUMBER OF SUBSCRIPTS EXCEEDS 31 BOUNDS
163	FORMAT NUMBER USED IN A PREVIOUS FORMAT STMT FORMATSAVER
164	TOO MANY CONSTANTS IN DATA PART OR IV LIST DATA
165	INIT.VALUE/EQUIV. TABLE OVERFLOW: INCR. MAXEQUIV EQUIVALENCE
166	TYPE DECLARATOR EXPECTED IMPLICIT
167	REAL OR INTEGER ARITHMETIC EXPRESSION REQUIRED DOASEEK, RETURN
168	ZERO OR NEGATIVE NON-STANDARD RETURN NOT ALLOWED RETURN

NO.	ERROR MESSAGE AND COMPILER PROCEDURES
169	NON STANDARD RETURN NOT ALLOWED IN MAIN PROGRAM RETURNS
170	...UNKNOWN ERROR ... BUILDPGMDESC, CLEANOUTINFO, WRAPUP NOTE If error message 170 occurs, program reconfiguration is necessary.
171	DO PARAMETERS MUST BE CONSTANT IN DATA STATEMENT TRANSCLASS
172	LENGTHS OF LIST AND DATA SET DO NOT AGREE
173	FILE NUMBER OR NAME EXPECTED LABELR
174	FILE OPTION NOT RECOGNIZED BUILDFIB
175	MULTIPLE FILE DEFINITION FILECHECK
176	RETURN NOT ALLOWED BLOCK IN DATA SUB PGM-IGNORED RETURN
177	NAMELIST ID PREVIOUSLY DEFINED NAMEL
178	FORMAL PARAM ILLEGAL AS NAMELIST ID OR ELEMENT NAMEL

FORTRAN COMPILER ERROR MESSAGES

NO.	ERROR MESSAGE AND COMPILER PROCEDURES
179	GLOBAL NAMELIST ELEMENT ILLEGAL WITH SEPARATE NAMEL
180	IMPLICIT STMT NOT SECOND STMT IN THE SUBPROGRAM IMPLICIT
181	IMPLICIT STMT NOT FIRST STMT IN MAIN PROGRAM IMPLICIT
182	FUNCTION TYPE INCONSISTENT WITH FUNCTION USAGE FCNREF, IMPLICIT, LOOKFORINTRINSIC
183	EMPTY EDITING SPECIFICATION PART NOT ALLOWED MOVENUM
184	ALREADY INITIALIZED DATA
185	SUBSCRIPT EXCEEDS ARRAY LENGTH DATA, TRANSCLASS
186	FORMAL PARAMETERS MAY NOT BE INITIALIZED DATA, TRANSCLASS
187	UNRECOGNIZED COMPILER OPTION
188	DATA OR INITIAL VAL APPEARS AFTER 1ST REFERENCE DATA, TRANSCLASS
189	STRING TRUNCATED - PRIMARY REQUIRED DATA

NO.	ERROR MESSAGE AND COMPILER PROCEDURES
190	LIMIT EXCEEDED FOR DATA AND INITIAL VALUE LIST DATA
191	BLOCK DATA ILLEGAL WITH SEPARATE OPTION BLOCKDATA
192	INVALID $ CARD OPTION --- IGNORED EMITLABELDESC
193	LABEL REQD ON STMT FOLLOWING GOTO & ARITH IF ARITHIFII, GOTOII
194	MISSING STATEMENT AFTER LOGICAL IF PRESCANNER2
195	STMT FUNCTION DEF FOLLOWS EXECUTABLE STATEMENT STMTFUN
196	NO LABEL FORMAL PARAM-NONSTANDARD RETURN ILLEGAL RETURN
197	OBSOLETE FILE OPTION IGNORED-USE LOCK STATEMENT BUILDFIB
198	ILLEGAL DEBUG STATEMENT LABELR
199	ALREADY BEING MONITORED MONITORIT
200	THIS CONSTRUCT IS NOT IMPLEMENTED FOR OPT>0 PARAMETERSONG, EMITPAUSEORZIP, EQUIVTWOARRAYS, FORMALPP

FORTRAN COMPILER ERROR MESSAGES

NO.	ERROR MESSAGE AND COMPILER PROCEDURES
201	OBSOLETE FILE OPTION --- IGNORED
202	DUMP STMT MUST PRECEDE LABEL WHERE DUMP OCCURS DUMPIT
203	MISSING DECIMAL POINT IN D,E,F,G FORMAT SPEC PHRASER
204	ILLEGAL REPEAT COUNT IN FORMAT SPECIFICATION WIDTHANDDECIMAL
205	STATEMENT IS REQUIRED AFTER EXPRESSION IN LOG IF
206	FORMAT EXCEEDS 24100 CHARACTERS GETNUM
207	ILLEGAL NUMBER OF CHARACTERS PER WORD PROCESSNAME
208	ILLEGAL STATEMENT SPECIFIER IN TRACE STATEMENT
209	INVALID CHARACTER SCANR
210	STMT. FUNCTION DEFINITIONS CANNOT BE RECURSIVE EXPANDSTATEMENTFUNCTION
211	ILLEGAL SIZE SPECIFICATION OPTIONLAST
212	ARITHMETIC OPERAND OR LEFT PARENTHESIS EXPECTED EXPR

NO.	ERROR MESSAGE AND COMPILER PROCEDURES
213	STATIC FORMAT REPEAT PART=0 --- PHRASE IGNORED
214	REDUNDANT FREEFIELD NAME PART
215	FREEFIELD LIST ELEMENT NAME VALID ON OUTPUT ONLY FORMATTEDIO
216	COMPILATION ABORTED DUE TO PARITY OR DATA ERROR

FORTRAN FORMAT ERROR MESSAGES

The following is an explanation of format error numbers. The meanings of the various error numbers pertaining to input are as follows:

NO.	ERROR MESSAGE
201	INVALID DATA FOR "I" FORMAT
202	INVALID DATA FOR "O" FORMAT
203	INVALID DATA FOR "Z" FORMAT
204	INVALID DATA FOR "L" FORMAT
205	INVALID DATA FOR "E", "F", OR "D" FORMAT
206	MISSING DECIMAL PLACES SPECIFICATION - "V" FORMAT
207	MISSING WIDTH SPECIFICATION - "V" FORMAT
208	INVALID PHRASE CODE - "V" FORMAT
209	INPUT VALUE TOO LARGE FOR LIST ELEMENT
210	ATTEMPTED TO EXCEED THE END OF THE INPUT RECORD

The meanings of the various error numbers pertaining to output are as follows:

NO.	ERROR MESSAGE
102	FORMAT WAS V SPECIFIER, AND LIST ELEMENT DID NOT PRODUCE AN F,D,E,G,Z,A,O,C,J,I,L,P,X, OR T. NOTE THAT ONLY THE RIGHT-MOST CHARACTER OF THE LIST ELEMENT IS USED. IF THE RIGHT-MOST CHARACTER IS BLANK, THEN THE LEFT-MOST CHARACTER IS USED. THE LIST ELEMENT MUST BE SINGLE PRECISION.
103	FORMAT WAS V SPECIFIER OF THE FORM RV, AND THE RESULTANT SPECIFIER NEEDED A FIELD WIDTH; FOR EXAMPLE, 2V => 2I.
104	FORMAT WAS V SPECIFIER OF THE FORM RV, AND THE RESULTANT SPECIFIER NEEDED A FIELD WIDTH AND DECIMAL PLACES; E.G., 3V => 3E.
105	FORMAT WAS V SPECIFIER OF THE FORM RVW, AND THE RESULTANT SPECIFIER NEEDED DECIMAL PLACES; E.G., 2V * => 2F6.

FORTRAN FORMAT ERROR MESSAGES

NO.	ERROR MESSAGE
106	FORMAT SPECIFIER EVALUATED TO FW.D FORM, AND D < 0.
107	FORMAT SPECIFIER EVALUATED TO EW.D OR DW.D, AND D < 1.
108	FORMAT WAS V SPECIFIER, AND LIST ELEMENT USED TO EVALUATE SPECIFIER TYPE (A,C,...ETC.) WAS DOUBLE PRECISION. ONLY SINGLE PRECISION IS ALLOWED.
109	FORMAT SPECIFIER EVALUATED TO GW, AND CORRESPONDING LIST ELEMENT WAS NEITHER OF TYPE INTEGER NOR TYPE LOGICAL (EXPRESSIONS OF TYPE INTEGER OR LOGICAL ARE EDITED UNDER GW.D AS IW OR LW, RESPECTIVELY). THEREFORE, THE DECIMAL PLACES ARE CONSIDERED MISSING.
110	A LONG STRING HAS BEEN USED WITH A NUMERIC EDITING FORMAT.
111	FORMAT SPECIFIER EVALUATED TO GW.D AND GW.D LOGIC CHOSE EW.D BUT D < 1.
112	FORMAT STATEMENT HAD NO FORMAT SPECIFIERS REQUIRING LIST ELEMENTS, AND FORMAT WAS USED WITH A LIST.
113	FORMAT SPECIFIER EVALUATED TO EW.D OR DW.D, AND W LEQ D.
114	DYNAMIC W OR D PART OF FORMAT SPECIFIER EVALUATED TO A VALUE GREATER THAN THE MAXIMUM INTEGER ALLOWED, 549755813887.
115	DYNAMIC PART OF FORMAT SPECIFIER EVALUATED TO A VALUE GREATER THAN THE MAXIMUM REAL ALLOWED, 4.31359146673*10**68.
116	ATTEMPTED RECURSION: EVALUATION OF A LIST ELEMENT CAUSED A READ/WRITE/CLOSE ON THE CURRENT FILE.
117	RECORD OVERFLOW: ATTEMPTED TO FORMAT BEYOND END-OF-RECORD.

The meanings of error numbers pertaining to run-time (object-time) formatting are as follows:

NO.	ERROR MESSAGE
500	LEFT PARENTHESIS EXPECTED
501	ILLEGAL FORMAT CHARACTER
502	INTEGER TOO LARGE -- TRUNCATED TO 65535
503	INTEGER CONSTANT EXPECTED
504	UNSPECIFIED FIELD WIDTH
505	DECIMAL FIELD NOT COMPATIBLE WITH FIELD WIDTH
506	MISSING DECIMAL POINT IN F,E,D,G FORMAT

NO.	ERROR MESSAGE
507	EMPTY EDITING SPECIFICATION
508	ILLEGAL REPEAT COUNT
509	ERROR IN HOLLERITH CHARACTER COUNT
510	ILLEGAL INCREMENT FOR T FORMAT
511	UNSPECIFIED DECIMAL FIELD
512	UNSPECIFIED SCALE FACTOR
513	UNMATCHED STRING-BRACKET CHARACTER
514	TOO MANY RIGHT PARENTHESES
515	COMMA EXPECTED

APPENDIX E

FORTRAN SYNTAX

This appendix contains a description of the B 6700/B 7700 FORTRAN language employing standard Backus-Naur metalinguistic notation. This description is provided as a reference tool and for language analysis purposes.

The numbers in the left-hand margin are definition numbers, while the numbers in the right-hand margin indicate the definitions in which the metalinguistic variables appearing on that line are defined.

1	<empty> ::=	
2	<space> ::= {character represented by a card column containing no punches}	
3	<octal digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7	
4	<digit> ::= <octal digit> | 8 | 9	3
5	<hex digit> ::= <digit> | A | B | C | D | E | F	4
6	<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | $	
7	<slash> ::= /	
8	<FORTRAN character> ::= <space> | <digit> | <letter> | = | + | - | , | * | / |) | (| $ | ' | . | > | <	2,4,6
9	<EBCDIC character> ::= <FORTRAN character> | [| ; | % |] | ≥ | ≤ | @ | : | {other characters in standard EBCDIC set}	8
10	<proper string character> ::= {any EBCDIC character except apostrophe} | {double apostrophe}	
11	<octal literal> ::= <octal digit> | <octal literal> <octal digit>	3
12	<integer literal> ::= <digit> | <integer literal><digit>	4
13	<decimal literal> ::= <integer literal>. | .<integer literal> | <integer literal>.<integer literal>	12
14	<hex literal> ::= <hex digit> | <hex literal> <hex digit>	5
15	<real literal> ::= <decimal literal><optional single exponent> | <integer literal><single exponent>	13,29 12,32
16	<double literal> ::= <decimal literal><double exponent> | <integer literal><double exponent>	13,33 12

FORTRAN SYNTAX

17	<complex literal> ::= (<real component>,<imaginary component>)	30,31
18	<logical literal> ::= .TRUE. | .FALSE.	
19	<string> ::= <Hollerith string> | <proper string literal>	20,21
20	<Hollerith string> ::= <integer literal> H <EBCDIC string>	12,35
21	<proper string literal> ::= '<proper string>'	36
22	<label> ::= <digit> |<label><digit>	4
23	<integer number> ::= <optional sign><integer literal>	34,12
24	<real number> ::= <optional sign><real literal>	34,15
25	<unsigned number> ::= <integer literal> |<real literal> | <double literal> | <complex literal>	12,15 16,17
26	<number> ::= <optional sign><unsigned number>	34,25
27	<octal number> ::= O <optional sign> <octal literal>	34,11
28	<hex number> ::= Z <optional sign> <hex literal>	34,14
29	<optional single exponent> ::= <empty> | <single exponent>	32
30	<real component> ::= <real number> | <integer number>	24,23
31	<imaginary component> ::= <real number> | <integer number>	24,23
32	<single exponent> ::= E<integer number>	23
33	<double exponent> ::= D<integer number>	23
34	<optional sign> ::= <empty> | + | -	
35	<EBCDIC string> ::= <EBCDIC character> | <EBCDIC string> <EBCDIC character>	9
36	<proper string> ::= <proper string character> | <proper string> <proper string character>	10
37	<identifier> ::= <letter> | <identifier><alphanumeric character>	6,55
38	<variable> ::= <simple variable> |<array element>	42,43
39	<function designator> ::= <external function designator> | <statement function designator>	40 41
40	<external function designator> ::= <function name> (<actual argument list>)	52 46
41	<statement function designator> ::= <function name> (<sf actual argument list>)	52
42	<simple variable> ::= <variable name>	48
43	<array element> ::= <array name>(<subscript list>)	51,44
44	<subscript list> ::= <subscript expression> |<subscript list>, <subscript expression>	45
45	<subscript expression> ::= <arithmetic expression>	57
46	<actual argument list> ::= <actual argument> | <actual argument list>,<actual argument>	47
47	<actual argument> ::= <expression> |<array name> | <string>|<function name>|<subroutine name>| <entry name> |&<label>	56,51 19,52,53 54,22

48 <sf actual argument list> ::= <sf actual argument> | 48
<sf actual argument list>, <sf actual argument>

49 <sf actual argument> ::= <expression> 56

50 <variable name> ::= <identifier> 37

51 <array name> ::= <identifier> 37

52 <function name> ::= <identifier> 37

53 <subroutine name> ::= <identifier> 37

54 <entry name> ::= <identifier> 37

55 <alphanumeric character> ::= <letter> |<digit> 6,4

56 <expression> ::= <arithmetic expression> | <logical expression> 57,61

57 <arithmetic expression> ::= <optional sign><term> | 34,58
<arithmetic expression><adding operator><term> 69

58 <term> ::= <factor> | <term><multiplying operator><factor> 59,58,68

59 <factor> ::= <primary> |<factor>**<primary> 60

60 <primary> ::= <unsigned number> |<variable>| 25,38
<function designator> |(<arithmetic expression>) | <proper 39,57
string literal> | <Hollerith string> 21

61 <logical expression> ::= <logical term> | <logical expression>.OR. 62
<logical term>

62 <logical term> ::= <logical factor> | <logical term>.AND. 63
<logical factor>

63 <logical factor> ::= <logical primary> | .NOT.<logical primary>| 64
<relation> 65

64 <logical primary> ::= <logical literal> | <logical variable> | 18,66
<logical function designator> | 67
(<logical expression>) 61

65 <relation> ::= <arithmetic expression><relational operator> 57,70
<arithmetic expression>

66 <logical variable> ::= <variable> 38

67 <logical function designator> ::= <function designator> 39

68 <multiplying operator> ::= * |<slash> 7

69 <adding operator> ::= + | -

70 <relational operator> ::= .LT.| .LE. |.EQ. |.NE. |.GT. |
.GE.| .IS.| >|<

71 <statement> ::= <label part> <executable statement>| 72,73
<nonexecutable statement>| <label> <format statement> 74,22,178

72 <label part> ::= <empty> | <label> 1,22

73 <executable statement> ::= <assignment statement> | <control 75
statement> |<input/output statement> 76,79

74 <nonexecutable statement> ::= <specification statement>| 82
<data statement> |<namelist statement> | <format 196,191
statement>| <function statement> | <statement function 178,215
declaration> | <subroutine statement>| <block data 221,216
statement>| <end statement> 214,223

FORTRAN SYNTAX

75	<assignment statement> ::= <variable>=<expression> | <assign statement>	38,56 84
76	<control statement> ::= <type-1 control statement> | <type-2 control statement>	77 78
77	<type-1 control statement> ::= <go to statement> |<arithmetic if statement> | <call statement>| <return statement>| <continue statement> |<stop statement> | <call exit statement>| <pause statement> |<zip statement>	83 91,89,90 95,97,98 96,99
78	<type-2 control statement> ::= <logical if statement> |<do statement>	92 94
79	<input/output statement> ::= <read statement> | <output statement> | <auxiliary I/O statement>	149 80,81
80	<output statement> ::= <write statement> | <print statement> | <punch statement>	150,154 153
81	<auxiliary I/O statement> ::= <backspace statement> | <close statement> | <endfile statement>| <find statement> | <lock statement> | <purge statement> | <rewind statement>	158 159,156,155 160,161,157
82	<specification statement> ::= <type statement>| <common state-ment> | <dimension statement> | <equivalence statement> | <external statement> |<implicit type statement>	113 114,112,115 116,121
83	<go to statement> ::= <unconditional go to statement>| <assigned go to statement> |<computed go to statement>	85 86,88
84	<assign statement> ::= ASSIGN<label>TO<variable>	22,38
85	<unconditional go to statement> ::= GO TO<label>	22
86	<assigned go to statement> ::= GO TO<variable> <label part>	38,87
87	<label part> ::= <empty> | ,(<label list>)	1,101
88	<computed go to statement> ::= GO TO(<label list>), <arithmetic expression>	101 57
89	<call statement> ::= CALL<subroutine name> <optional actual argument part>	53 102
90	<return statement> ::= RETURN<optional return number>	103
91	<arithemtic if statement> ::= IF(<arithmetic expression>) <label>,<label>,<label>	57 22
92	<logical if statement> ::= IF(<logical expression>) <logical executable statement>	61 93
93	<logical executable statement> ::= <assignment statement>| <input/output statement> | <type-1 control statement>	75 79,77
94	<do statement> ::= DO<label><do control>	22,104
95	<continue statement> ::= CONTINUE	
96	<pause statement> ::= PAUSE<optional pause number>	110
97	<stop statement> ::= STOP<optional stop number>	111
98	<call exit statement> ::= CALL EXIT	
99	<zip statement> ::= ZIP WITH <control source>	99
100	<control source> ::= <array name> |<file designator>	51,162

101	<label list> ::= <label> |<label list>,<label>	22
102	<optional actual argument part> ::= <empty> |(<actual argument list>)	1 46
103	<optional return number> ::= <empty> | <arithmetic expression>	1,57
104	<do control> ::= <do index>=<initial parameter>,<terminal parameter> <optional do increment>	105,106 107,108
105	<do index> ::= <simple variable>	42
106	<initial parameter> ::= <arithmetic expression>	57
107	<terminal parameter> ::= <arithmetic expression>	57
108	<optional do increment> ::= <empty> | ,<incrementation parameter>	1 109
109	<incrementation parameter> ::= <arithmetic expression>	57
110	<optional pause number> ::= <empty> |<integer literal> | <proper string literal>	1,12 21
111	<optional stop number> ::= <empty>| <integer literal>| <proper string literal>	1,12 21
112	<dimension statement> ::= DIMENSION<dimension list>	131
113	<type statement> ::= <type declarator> <declaration list>	118 119
114	<common statement> ::= COMMON<optional block identifier> <common list><optional block declaration part>	136 138,140
115	<equivalence statement> ::= EQUIVALENCE<equivalence group list>	143
116	<external statement> ::= EXTERNAL<external list>	147
117	<type> ::= INTEGER | REAL | DOUBLE PRECISION | COMPLEX |LOGICAL	
118	<type declarator> ::= <type> |<type> * <integer literal>	117,12
119	<declaration list> ::= <declaration element>| <declaration list>, <declaration element>	130
121	<implicit type statement> ::= IMPLICIT <type list>	122
122	<type list> ::= <initial character declarator>| <type list>,<initial character declarator>	123
123	<initial character declarator> ::= <type declarator> (<initial character list>)	118 124
124	<initial character list> ::= <letter> |<letter>-<letter> | <initial character list>, <letter>| <initial character list>, <letter>-<letter>	6
125	<initial value list> ::= <initial value>| <initial value list>, <initial value>	126
126	<initial value> ::= <repeat constant> <initial value element>	128,127
127	<initial value element> ::= <number> |<octal number> | <hex number> | <string> | T|F	26,27 28,19
128	<repeat constant> ::= <empty> | <integer literal> *	1,12

FORTRAN SYNTAX

129 <declaration ID> ::= <variable name> | <function name> | <array name> <optional array bounds> | <entry name> — 50,52 51,133,54

130 <declaration element> ::= <declaration ID> <size part> | <declaration list> <slash> <initial value list> <slash> — 129,230 119,125

131 <dimension list> ::= <dimension element> | <dimension list>, <dimension element> — 132

132 <dimension element> ::= <array name>(<array bounds list>) — 51,134

133 <optional array bounds> ::= <empty> | (<array bounds list>) — 1,134

134 <array bounds list> ::= <array bound> | <array bounds list>, <array bound> — 135

135 <array bound> ::= <integer literal> | <variable name> | <real literal> — 12,48 15

136 <optional block identifier> ::= <empty> | <slash><block name><slash> — 1,7 137

137 <block name> ::= <empty> | <identifier> — 1,37

138 <common list> ::= <common element> | <common list>, <common element> — 139

139 <common element> ::= <variable name> | <array name> <optional array bounds> — 50,51 133

140 <optional block declaration part> ::= <empty> | <block declaration part> — 1 141

141 <block declaration part> ::= <block declaration> | <block declaration part><block declaration> — 142

142 <block declaration> ::= <slash> <block name> <slash> <common list> — 7,137 138

143 <equivalence group list> ::= <equivalence group> | <equivalence group list>, <equivalence group> — 144

144 <equivalence group> ::= (<equivalence list>) — 145

145 <equivalence list> ::= <equivalence element> | <equivalence list>, <equivalence element> — 146

146 <equivalence element> ::= <variable> | <array name> — 38,51

147 <external list> ::= <external element> | <external list>, <external element> — 148

148 <external element> ::= <subroutine name> | <entry name> | <function name> — 53,54 52

149 <read statement> ::= READ (<file designator> <optional record specifier> <optional format specifier> <optional result clause list>) <optional I/O list> | READ<format specifier> | READ<format specifier>,<I/O list> — 162 163,166 168,175 167,176

150 <write statement> ::= <formatted write statement> | <unformatted write statement> — 151 152

151 <formatted write statement> ::= WRITE (<file designator> <optional record specifier>, <format specifier> <optional result clause list>) <optional I/O list> — 162 163,167 168,175

152 <unformatted write statement> ::= WRITE (<file designator> <optional record specifier> <optional result clause list>) <I/O list> — 162 163,168 176

153 <punch statement> ::= PUNCH<format specifier> | PUNCH <format specifier>, <I/O list> — 167, 176

154 <print statement> ::= PRINT<format specifier> | PRINT <format specifier>,<I/O list> — 167, 176

155 <find statement> ::= FIND (<file designator><record specifier>) — 162,164

156 <endfile statement> ::= ENDFILE<file designator> — 162

157 <rewind statement> ::= REWIND<file designaotr> — 162

158 <backspace statement ::= BACKSPACE<file designator> — 162

159 <close statement> ::= CLOSE<file designator> — 162

160 <lock statement> ::= LOCK<file designator> — 162

161 <purge statement> ::= PURGE<file designator> — 162

162 <file designator> ::= <arithmetic expression> — 57

163 <optional record specifier> ::= <empty> | <record specifier> — 1,164

164 <record specifier> ::= '<record number> | =<record number> — 165

165 <record number> ::= <arithmetic expression> — 57

166 <optional format specifier> ::= <empty> | ,<format specifier> — 1,167

167 <format specifier> ::= <label> | <array name> | <namelist name> — 22,51,193

168 <optional result clause list> ::= <empty> | ,<result clause list> — 1, 168

169 <result clause list> ::= <result clause> | <result clause list>, <result clause> — 170

170 <result clause> ::= <parity error action> | <end-of-file action> | <invalid format action> | <result descriptor clause> — 171, 172,173,174

171 <parity error action> ::= ERR = <label> — 22

172 <end-of-file action> ::= END = <label> — 22

173 <invalid format action> ::= DATA = <label> — 22

174 <result descriptor clause> ::= RESULT = <variable> — 38

175 <optional I/O list> ::= <empty> | <I/O list> — 1,176

176 <I/O list> ::= <I/O list element> | <I/O list>, <I/O list element> — 177

177 <I/O list element> ::= <variable> | <array name> | (<I/O list>, <do control>) | <arithmetic expression> — 38,51, 176,104,57

178 <format statement> ::= FORMAT (<editing specifications>) — 179

179 <editing specifications> ::= <editing segment> | <editing specifications> <record skip> <editing specifications> | <editing specifications> <record skip> <editing segment> — 180, 181

180 <editing segment> ::= <editing phrase> | <editing segment>, <editing phrase> | <repeat part>(<editing specifications>) | <editing segment>, <repeat part>(<editing specifications>) — 182, 185,179

181 <record skip> ::= <slash> | <record skip><slash> — 7

FORTRAN SYNTAX

182	<editing phrase> ::= <scaling phrase><repeat part><defg> <field width>.<decimal places> | <repeat part> <aigloz> <field width>| T<character position> |<repeat part> X | <string>	183,185 186,187,190 188 19
183	<scaling phrase> ::= <empty>| <scale factor>P	1,184
184	<scale factor> ::= <integer literal> | *	12
185	<repeat part> ::= <empty> | <integer literal> | *	1,12
186	<field width> ::= <integer literal> | *	12
187	<decimal places> ::= <integer literal>| *	12
188	<character position> ::= <integer literal>| *	12
189	<defg> ::= D| E | F |G | V	
190	<aigloz> ::= A |I| G |L |O | Z |V| C | J	
191	<namelist statement> ::= NAMELIST<namelist list>	192
192	<namelist list> ::= <slash> <namelist name> <slash> <namelist> | <namelist list> <slash> <namelist name> <slash> <namelist>	7,193 194
193	<namelist name> ::= <identifier>	37
194	<namelist> ::= <namelist element> |<namelist>, <namelist element>	195
195	<namelist element> ::= <variable name>|<array name>	48,51
196	<data statement> ::= DATA<data set list>	197
197	<data set list> ::= <data set> <data set list>,<data set>	198
198	<data set> ::= <data I/O list>/<data part list>/	199,205
199	<data I/O list> ::= <data I/O list element>|<data I/O list>, <data I/O list element>	200
200	<data I/O list element> ::= <variable>| <array name>| (<data I/O list>, <data do control>)	38,51 199,201
201	<data do control> ::= <do index> = <constant initial parameter>, <constant terminal parameter> <optional constant incrementation parameter>	105 202,203 204
202	<constant initial parameter> ::= <integer literal> | <real literal>	12 15
203	<constant terminal parameter ::= <integer literal> | <real literal>	12 15
204	<optional constant incrementation parameter> ::= <empty> | ,<integer literal> |, <real literal>	1 12,15
205	<data part list> ::= <data part> |<data part list>, <data part>	206
206	<data part> ::= <data value>| <integer literal>*<data value> | <string> | <integer literal>*<string>	207,12 19
207	<data value> ::= <number> | <octal number> | <hex number>| <logical literal>| T | F	26,27,28 18
208	<program> ::= <main program> |<program><subprogram> | <subprogram><program>	209,210
209	<main program> ::= <main program body>	217

210	<subprogram> ::= <block data subprogram> | <function subprogram> | <subroutine subprogram>	211,212 213
211	<block data subprogram> ::= <block data statement> <block data subprogram body>	214 219
212	<function subprogram> ::= <function statement><subprogram body>	215 218
213	<subroutine subprogram> ::= <subroutine statement><subprogram body>	216 218
214	<block data statement> ::= BLOCK DATA	
215	<function statement> ::= <optional type declarator> FUNCTION <function name>(<dummy argument list>)	224 52,226
216	<subroutine statement> ::= SUBROUTINE<subroutine name> <optional dummy argument part>	53 225
217	<main program body> ::= <end statement> | <statement> <main program body>	223,71
218	<subprogram body> ::= <end statement> | <statement> <subprogram body> | <entry statement><subprogram body>	223,71 222
219	<block data subprogram body> ::= <end statement> | <block data subprogram statement> <block data subprogram body>	223 220
220	<block data subprogram statement> ::= <specification statement> | <data statement>	82,196
221	<statement function declaration> ::= <function name> (<sf dummy argument list>) = <expression>	52 227,56
222	<entry statement> ::= ENTRY <entry name> <optional dummy argument part>	54 225
223	<end statement> ::= END	
224	<optional type declarator> ::= <empty> | <type declarator>	1,118
225	<optional dummy argument part> ::= <empty> | (<dummy argument list>)	1 226
226	<dummy argument list> ::= <dummy argument> | <dummy argument list>,<dummy argument>	228
227	<sf dummy argument list> ::= <sf dummy argument> | <sf dummy argument list>, <sf dummy argument>	229
228	<dummy argument> ::= <identifier> | * | <slash> <identifier> <slash>	37,7
229	<sf dummy argument> ::= <identifier>	37
230	<size part> ::= <empty> | * <integer literal>	1,12

INDEX

Item	Page
A format	13-2
action specifier	12-5
additional language capabilities	16-1
alphabetic characters	3-2
AREA	B-7
arithmetic assignment statement	9-1
arithmetic expressions	6-2
arithmetic IF statement	8-10
arithmetic operators	6-2
arrays	5-2
declarators	5-3, 7-2, 7-4
elements	5-2, 5-5
FORMAT	13-21
handling	8-2, 11-1
name	5-3
storage	11-1, 11-7
ZIP WITH	16-5
ASCII option	19-6
assigned GO TO	8-8, 9-3
assignment statements	9-1
arithmetic	9-1
GO TO	9-3
logical	9-2
AUTOBIND option	19-7
BACKSPACE statement	12-18
batch facility	18-19
deck set-up	18-19
restrictions	18-21
use of	18-20
BCD	19-8
characters	3-4
input	3-3
BCD option	19-8
BCL	19-8
characters	3-3
input	3-3, 4-12
BCL option	19-8
BIGOPTJOB option	19-27
BINDER option	19-8
BLOCK DATA	15-9
BLOCKING	B-8
BUFFER	B-8
buffers, use of	12-10
B5500 option	19-8
B5700 option	19-9
B7700 option	19-9

INDEX

Item	Page
C format	13-3
CALL statement	8-1
array handling	8-2
subroutine returns	8-4
CALL EXIT statement	8-4
CANDE, compilations through	18-22
card files	12-30
card punch files	12-30
carriage control, FORMAT	13-20
CHANGE statement	12-19
character card codes	3-4
character set	3-1
characters	3-1
BCD	3-4
BCL	3-3
EBCDIC	3-2
FORTRAN	3-2
CHARS option	19-9
CHECK option	19-9
CLOSE statement	12-20
CODE option	19-10
COMMON	2-3
block	7-4, 7-5, 7-7
names	7-4
statement	2-3, 7-4, 11-14
COMMON storage, elements of	11-14
storage assignments	7-5, 11-14
compatibility, language	18-22
ANSI FORTRAN	18-25
B 3500 FORTRAN	18-23
FORTRAN IV-H	18-22
other FORTRAN languages	18-25
compilation, input card file	18-12
go compile	18-16
library compile	18-15
library and go compile	18-18
syntax compile	18-13
compilations through CANDE	18-22
compilations, with NAMELIST	14-5
compiler	18-1
control cards	18-13, 19-1, 20-3
control statements	19-1
error messages	C-1
files	18-4
options	19-1
compiler files	18-4
input	18-6
output	18-6
table	18-8
compiler options	19-1
generation options	19-27
option actions	19-2
options	19-5
COMPLEX	4-6
constant	4-6
type operand	6-3
variable	5-1, 11-3

INDEX

Item	Page
computed GO TO	8-8
CONCAT intrinsic	15-13
constant	4-1
complex	4-6
double precision	4-4
hexadecimal	4-8
integer	4-2
logical	4-10
octal	4-7
real (floating-point)	4-2
CONTINUE statement	8-4
control cards, compiler	18-13, 18-14, 18-15, 18-16, 18-18
control statements	8-1
CALL	8-1
CALL EXIT	8-4
CONTINUE	8-4
DO	8-5
GO TO	8-7
IF	8-9
PAUSE	8-11
RETURN	8-12
STOP	8-14
ZIP WITH	8-15
CTIME intrinsic	15-14
CYLINDERMODE	B-9
D format	13-4
DATA statements	2-3, 10-1
considerations	10-4
data transfer, internal	12-11, 12-15
data word	1-4
DBLTOSNGL option	19-10
DEBUG option	19-10
DEBUG statement	17-1
DUMP	17-3
MONITOR	17-1
PROGRAMDUMP	17-4
STATISTICS	17-5
TRACE	17-6
debugging and timing aids	17-1
decimal digits	3-1
declaration cards, file	B-1
default file specifications	B-14
digits	3-1
decimal	3-1
hexadecimal	3-2
octal	3-1
DIMENSION statement	2-3, 7-5
disk files	12-30
DO statement	8-5
nesting	8-7
parameter alteration	8-7
DOUBLE PRECISION	4-4
constant	4-4, 10-3
type operand	6-3
variable	5-1, 11-4
DUMP, DEBUG statement	17-3

INDEX

Item	Page
E format	13-5
EBCDIC characters	3-2
EBCDIC input	3-1, 4-11
END statement	2-2
END option	19-10
ENDFILE statement	12-21
ENTRY option	19-10
ENTRY statement	15-7
dummy argument list	15-8
function subprograms	15-8
name	15-8
subroutine subprograms	15-8
EQUIVALENCE data items	11-10
multiple locations	11-11
single locations	11-10
storage	11-10
EQUIVALENCE statement	2-3, 7-6
ERRLIST option	19-10
error messages	C-1, D-1
executable statements	6-6
expressions	6-1
arithmetic	6-2
relational	6-4
types of	6-3
expressions and statements	6-1
EXTERNAL statement	7-7
F format	13-6
FASTGRAPH option	19-27
field separators, FORMAT	13-18
file declaration cards	B-1
file designators	12-1
file format, ZIP WITH	16-8
construction	16-9
use of	16-11
file handling, capabilities	12-29
card reader	12-30
card punch	12-30
desk	12-30
line printer	12-29
tape	12-30
file specifications, default	B-14
files, compiler	18-4
input	18-6
output	18-6
table	18-8
files referenced	12-9, 12-13, 12-16
FIND statement	12-22
error handling	12-23
format considerations, PRINT	12-17
format error messages	D-1
format modifiers	13-17
K format	13-17
$ format	13-17
format specifications and statements	13-1
format specifications	13-1
A format	13-2

INDEX

Item	Page
C format	13-3
D format	13-4
E format	13-5
F format	13-6
G format	13-7
H format	13-8
I format	13-9
J format	13-10
L format	13-11
O format	13-12
P format	13-13
T format	13-14
U format	13-15
V format	13-15
X format	13-15
Z format	13-15
FORMAT statement	2-3, 13-17
arrays	13-21
carriage control	13-20
complex I/O list elements	13-18
format field separators	13-18
record fields	13-18
repeat count	13-19
run-time format editing	13-19
FORTRAN	1-2
characters	3-2
concepts	1-4
language B 6700/B 7700	1-2
FORTRAN compiler	18-1
files	18-4
system requirements	18-1
options	19-5
formatted READ	12-10
formatted WRITE	12-14
FREE option	19-11
free-format, INPUT/OUTPUT	12-28
FREENEW option	19-11
FREETAPE option	19-11
FUNCTION	15-3
name	15-4
statement	15-3
subprogram	15-3
use of	15-6
G format	13-7
generation options, compiler	19-27
GETFIRSTDOLLARCARDFROMSPO option	19-27
go, compile for	18-16
GO TO statement	8-7
assigned	8-8, 9-3
computed	8-8
unconditional	8-8
GRAPH option	19-11
H format	13-8
hardware requirements	18-1
hexadecimal constant	4-8, 10-3

INDEX

Item	Page
hexadecimal digits	3-2
Hollerith string	4-11
I format	13-9
IF statement	8-9
arithmetic	8-10
logical	8-10
IMPLICIT statement	2-3
initial character list	7-8
INCLNEW option	19-11
INCLUDE option	19-12
initial character list, IMPLICIT statement	7-8
initial value list	7-3, 10-2
conversions during assignment	10-4
input	3-1
BCD	3-3
BCL	3-3
EBCDIC	3-1
input card file, compilation	18-12
go compile	18-13
library compile	18-15
library and go compile	18-18
syntax compile	18-13
input, NAMELIST	14-2
INPUT/OUTPUT statements	12-1
INQUIRE statement	12-23
INSTALLATION option	19-14
INTEGER	4-2
constants (ICB)	4-2
type operand	6-3
variable	5-1, 11-2
INTERCHANGE	B-9
interface	18-1
installation/compiler	18-1
user/compiler	18-3
I/O lists	12-4
I/O list elements, FORMAT	13-18
IOTIME option	19-15
I/O statements	12-8
READ	12-8
WRITE	12-12
PRINT	12-15
PUNCH	12-17
BACKSPACE	12-18
CHANGE	12-19
CLOSE	12-20
ENDFILE	12-21
FIND	12-22
INQUIRE	12-23
LOCK	12-24
OPEN	12-25
PURGE	12-26
REWIND	12-27
internal data transfer	12-11, 12-15
intrinsics	15-10
CONCAT	15-13
CTIME	15-14

INDEX

Item	Page
RANDOM	15-14
TIME	15-15
J format	13-10
JOB option	19-15
K format	13-17
L format	13-11
language compatibility	18-22
FORTRAN IV, level H	18-22
B 3500 FORTRAN	18-23
ANSI FORTRAN	18-25
other FORTRAN languages	18-25
language description conventions	1-2
letters	3-2
LEVEL option	19-15
LIBRARY option	19-16
library, compile for	18-15
library and go, compile for	18-18
LIMIT option	19-16
LINEINFO option	19-16
line printer files	12-29
LINKWORD	B-9
LIST option	19-16
literals	4-1
non-numeric	4-11
numeric	4-1
LOCK	12-24
logical assignment statement	9-2
logical constant	4-10
logical expressions	6-4
construction of	6-5
logical IF	8-10
logical operators	6-2
logical variable	5-1, 11-6
LONG option	19-17
main program	2-2
maintenance of compiler software	18-2
MERGE option	19-17
MODELII option	19-27
MONITOR, (DEBUG) statement	17-1
MONITOR option	19-17
MULTI option	19-27
NAMELIST	14-1
input	14-2
output	14-5
separate compilations	14-5
NAMELIST statements and NAMELIST I/O	14-1
nesting, DO	8-7
NEW option	19-18
NEWSEGMENT option	19-18
NEWSEQERR option	19-19
NOBINDINFO option	19-19
non-executable statements	6-7

INDEX

Item	Page
non-numeric literal	4-11
non-standard RETURN	8-13
NOWARN option	19-19
numeric literal	4-1
O format	13-12
octal constant	4-7, 10-3
octal digits	3-1
OMIT option	19-19
OMITDEBUG option	19-20
OPEN statement	12-25
operands	6-3
operators	6-1
OPT option	2-3, 19-20, 20-1
optimization	2-3, 20-1
OPTLSS1 option	19-27
output, NAMELIST	14-5
OWN option	19-20
OWNARRAYS option	19-21
P format	13-13
PAGE option	19-21
parameter alteration, DO	8-7
PAUSE statement	8-11
presence of operators	6-2
PRINT statement	12-16
files referenced	12-16
format considerations	12-17
record access	12-17
PRINTER option	19-21
printer files	12-29
PRINTLIMIT option	19-22
process initiation, independent	16-4
PROCESSTIME option	19-22
PROGRAMDUMP, DEBUG	17-4
program files	12-1
program structure	2-1
program units	2-2, 16-1
proper string	4-11, 6-3
PUNCH statement	12-17
punch files	12-30
PURGE	12-26
RANDOM	B-9
RANDOM intrinsic	15-14
READ statement	12-8
files referenced	12-9
formatted	12-10
internal data transfer	12-11
record numbers	12-10
unformatted	12-11
READER option	19-22
REAL	4-2
constant	4-2
type operand	6-3
variable	5-1, 11-3
RECORD	B-10

INDEX

Item	Page
record access, PRINT	12-17
record fields, FORMAT	13-18
record numbers	12-10, 12-13
recursion	15-7
related documents	1-5
relational operators	6-2
repeat counts, FORMAT	13-19
result clause list	12-5, 12-8
result descriptor, assignment	12-7
RETURN statement	8-12
non-standard	8-13
standard	8-12
REWIND statement	12-27
run-time format editing, FORMAT	13-19
sample program	A-1
SAVE	B-10
SEGMENTATION option	19-22
SEPARATE option	19-22
separate compilation considerations	B-12
SEQ option	19-23
SEQERR option	19-24
SERIAL	B-10
SETOWNARRAYS option	19-28
simple variables, storage	11-2
COMPLEX	11-5
DOUBLE PRECISION	11-4
INTEGER	11-2
LOGICAL	11-6
REAL	11-3
SINGLE option	19-24
SINGLEBYDEFAULT option	19-28
SINGLEPACK	B-10
software requirements	18-2
source input format	2-1
exceptions	2-1
specification statements	7-1
COMMON	7-4
DIMENSION	7-5
EQUIVALENCE	7-6
EXTERNAL	7-7
IMPLICIT	7-8
type	7-1
STACK option	19-25
standard RETURN	8-12
statement functions	15-5
statement labels	6-8
statement ordering, rules of	2-2
statements	6-6
executable	6-6
non-executable	6-7
statements, specification	7-1
STATISTICS, DEBUG	17-5
STOP statement	8-14
storage allocations	11-1
arrays	11-7
COMMON	11-14

INDEX

Item	Page
EQUIVALENCEd data items	11-10
simple variables	11-2
string literals	4-1, 4-11, 4-13
structure, program	2-1
subprograms	15-1
BLOCK DATA	15-9
function	15-3
SUBROUTINES	15-1
subprograms and intrinsics	15-1
SUBROUTINE	15-1
names	15-1
statement	15-1
subprograms	15-1
use of	15-2
subroutines returns	8-4
SUPRS option	19-25
syntax	E-1
syntax, compile for	18-13
system requirements	18-1
T format	13-14
tape files	12-30
multi-file reel	12-31
single reel	12-30
TIME option	19-25
TIME intrinsic	15-15
timing aids	17-1
TRACE option	19-25
TRACE, DEBUG	17-6
type statements	7-1
array declarators	7-2
initial value list	7-3
optional size specification	7-3
U format	13-15
unconditional GO TO	8-8
unformatted READ	12-11
unformatted WRITE	12-14
UNIT	B-6
UNLABELED	B-11
V format	13-15
VALUE construct	6-6
variables	5-1
names	5-1
type of	5-1
variable lists, DATA	10-1
VECTORMODE option	19-25, 20-1
vector mode	20-4
VECTORMODEISALLOWED option	19-28
VOID option	19-26
VOIDT option	19-26
WRITE statement	12-12
file referenced	12-13
formatted	12-14

INDEX

Item	Page
internal data transfer	12-15
record numbers	12-13
unformatted	12-14
X format	13-15
XREF option	19-26
Z format	13-15
ZIP WITH statement	6-4, 6-5, 6-8, 8-15, 16-4, 16-8
$ format	13-17
$ option	19-26

BURROUGHS CORPORATION
DATA PROCESSING PUBLICATIONS
REMARKS FORM

TITLE: B 6700/B 7700 FORTRAN Reference Manual

FORM: 5000458
DATE: 2 September 1974

CHECK TYPE OF SUGGESTION:

☐ ADDITION ☐ DELETION ☐ REVISION ☐ ERROR

GENERAL COMMENTS AND/OR SUGGESTIONS FOR IMPROVEMENT OF PUBLICATION:

FROM: NAME _____ DATE _____
TITLE _____
COMPANY _____
ADDRESS _____
_____

cut along dotted line

Fold Along Dashed Line

Fold, Staple, And Mail

Postage Will Be Paid by Addressee

No Postage Stamp Necessary If Mailed in the United States

BUSINESS REPLY MAIL
First Class Permit No. 381; City of Industry, Calif. 91744

Burroughs Corporation
P. O. Box 1223
City of Industry, Calif. 91744

attn: Publications Department
Technical Information Organization

Fold, Staple, And Mail